Replay Debugging for the Datacenter

Gautam Altekar

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-216
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-216.html

December 1, 2012

Replay Debugging for the Datacenter

by

Gautam Deepak Altekar

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Koushik Sen
Professor Ray Larson

Spring 2011

Replay Debugging for the Datacenter

Abstract

Replay Debugging for the Datacenter

by

Gautam Deepak Altekar

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Debugging large-scale, data-intensive, distributed applications running in a datacenter ("datacenter applications") is complex and time-consuming. The key obstacle is non-deterministic failures—hard-to-reproduce program misbehaviors that are immune to traditional cyclic-debugging techniques. Datacenter applications are rife with such failures because they operate in highly non-deterministic environments: a typical setup employs thousands of nodes, spread across multiple datacenters, to process terabytes of data per day. In these environments, existing methods for debugging non-deterministic failures are of limited use. They either incur excessive production overheads or don't scale to multi-node, terabyte-scale processing.

To help remedy the situation, we have built a new deterministic replay tool. Our tool, called `DCR`, enables the reproduction and debugging of non-deterministic failures in production datacenter runs. The key observation behind `DCR` is that debugging does not always require a precise replica of the original datacenter run. Instead, it often suffices to produce some run that exhibits the original behavior of the *control-plane*—the most error-prone component of datacenter applications. `DCR` leverages this observation to relax the determinism guarantees offered by the system, and consequently, to address key requirements of production datacenter applications: lightweight recording of long-running programs, causally consistent replay of large-scale clusters, and out-of-the box operation with existing, real-world applications running on commodity multiprocessors.

To my family

Contents

Acknowledgments

I received much help leading up to this dissertation, for which I am very grateful.

First, I thank my fellow graduate students and lab mates: Jayanth Kannan, Karthik Lakshminarayanan, Dilip Joseph, Daekyeong Moon, Cheng Tien Ee, David Molnar, Ganesh Ananthanarayanan, Lisa Fowler, Michael Armbrust, Rodrigo Fonseca, George Porter, Matt Caesar, Blaine Nelson, Andrey Ermolinski, Ali Ghodsi, Matei Zaharia, Ari Rabkin, Byung-Gon Chun, Andrew Schultz, Ilya Bagrak, Paul Burstein, and others. You all were a fountain of insight.

The work presented in this dissertation includes contributions from several project collaborators. Dennis Geels's dissertation work on distributed replay set the foundations for this work. David Molnar's dissertation work on symbolic execution was the template for the symbolic execution engine presented herein. This work also contains some ideas jointly developed with Cristian Zamfir and George Candea, both of whom also provided fresh, outside perspective on the replay debugging problem. Many thanks to you all.

I thank my committee members Professor Ray Larson and Professor Koushik Sen for finding time for yet another dissertation committee. A special thanks goes to Professor Koushik Sen, whose work on concolic testing was the inspiration for the replay techniques developed for this dissertation.

My advisor Professor Ion Stoica had the vision and patience to support my research pursuits, even when they crossed the boundaries of his expertise. I am truly grateful to have worked with him, and for his guidance and advice, even though I may not have always taken it.

This dissertation would not have been possible without the support of my family. My grandparents nurtured me at a time when my parents were unable to. And my mother, Sita Altekar, provided me with the education and drive to aim big. A heartfelt thanks to all of you.

Finally, I thank my wife Shan for celebrating my successes, for encouraging me through my disappointments, for putting up with years of graduate school life, and for helping me see beyond it. I don't deserve you, but I'll keep you anyway.

Chapter 1

Introduction

The past decade has seen the rise of *datacenter applications*—large scale, distributed, data-intensive applications such as HDFS/GFS [26], HBase/Bigtable [12], and Hadoop/MapReduce [14]. These applications run on thousands of commodity nodes, spread across multiple datacenters, and process terabytes of data per day. More and more services that we use on a daily basis, such as Web search, e-mail, social networks (e.g., Facebook, Twitter), and video sharing rely on detacenter applications to meet their large-scale data processing needs. But as users and businesses grow more dependent on these hosted services, the datacenter applications that they rely on need to become more robust and available. To maintain high availability, it is critical to diagnose application failures and quickly debug them.

Unfortunately, debugging datacenter applications is hard for many reasons. A key obstacle is non-deterministic failures—hard-to-reproduce program misbehaviors that are immune to traditional cyclic-debugging techniques. These failures often manifest only in production runs and may take weeks to fully diagnose, hence draining the resources that could otherwise be devoted to developing novel features and services [47]. Another obstacle is the fact that the causality chain of a datacenter application failure may span multiple nodes and hence is more difficult to trace than the single-node case. Furthermore, datacenter applications typically operate on many terabytes of data every day and are required to maintain high throughput, which makes it hard to record what they do. Finally, these applications are usually part of complex software stacks that are used to provide 24x7 services, thus taking the application down for debugging is not an option. In sum, *effective tools for debugging failures in production datacenter systems are sorely needed.*

Developers presently use a range of methods for debugging application failures, but they all fall short in the datacenter environment. The widely-used approach of code instrumentation and logging requires either extensive instrumentation or foresight of the failure to be effective—neither of which are realistic in web-scale systems subject to unexpected production workloads. Automated testing, simulation, and source-code analysis tools [18, 33, 40] can find the errors underlying several failures before they occur, but the large state-spaces of datacenter systems hamper complete and/or precise results; some errors will inevitably

fall through to production. Finally, automated console-log analysis tools show promise in detecting anomalous events [49] and diagnosing failures [50], but the inferences they draw are fundamentally limited by the fidelity of developer-instrumented console logs.

In this dissertation, we show that *replay-debugging technology* (a.k.a, deterministic replay) can be used to debug failures in production datacenters. Briefly, a replay-debugger works by first capturing data from non-deterministic data sources such as the keyboard and network, and then substituting the captured data into subsequent re-executions of the same program. A cluster-wide deterministic replay solution is the natural option for debugging, as it offers developers the global view of the application "on a platter": by replaying failures deterministically, one can use a debugger to zoom in on various parts of the system and understand why the failure occurs. Without a cluster-wide replay capability, the developer would have to reason about global (i.e., distributed) invariants, which in turn can only be correctly evaluated at consistent snapshots in the distributed execution. Unfortunately, getting such consistent snapshots requires either a global clock (which is non-existent in clusters of commodity hardware) or expensive algorithms to capture consistent snapshots [11].

Developing a replay-debugging solution for the datacenter is harder than for a single node, due to the inherent runtime overheads required to do record-replay. First, these applications are typically data-intensive, as the volume of data they need to process increases proportionally with the size of the system (i.e., its capacity), the power of individual nodes (i.e., more cores means more data flowing through), and ultimately with the success of the business. Recording such large volumes of data is impractical. A second reason is the abundance of sources of non-determinism that must be captured, including the most notorious kind—multiprocessor data races. Some data races are bugs, while others are benign (intentionally placed for performance reasons). Regardless, the behavior of modern programs depend critically on the outcomes of these races. Unfortunately, capturing data races requires either expensive instrumentation or specialized hardware unavailable in datacenter nodes. A third challenge is of a non-technical nature: as economies of scale are driving the adoption of commodity clusters, the tolerable runtime overhead actually decreases—making up for a 50% throughput drop requires provisioning twice more machines; 50% of a 10,000-node cluster is a lot more expensive than 50% of a 100-node cluster. When operating large clusters, it actually becomes cheaper to hire more engineers to debug problems than to buy more machines to tolerate the runtime overhead resulting from a record-replay system.

Existing work in distributed system debugging does not offer solutions to these challenges. Systems like Friday [24] and WiDS [37] address distributed replay, but do not address the data intensive aspect, and therefore they are not suitable for datacenters. No existing system can replay at this scale and magnitude. If we are to cast off our continued reliance on humans to solve routine debugging tasks, we must devise smarter, more scalable debugging tools that aid developers in quickly debugging problems.

1.1 Requirements

Many replay debugging systems have been built over the years and experience indicates that they are invaluable in reasoning about non-deterministic failures [6, 10, 17, 24, 25, 36, 37, 39, 43, 52]. However, no existing system meets the following unique demands of the datacenter environment.

Always-On Operation. The system must be on at all times during production so that arbitrary segments of production runs may be replay-debugged at a later time.

In the datacenter, supporting always-on operation is difficult. The system should have minimal impact on production throughput (less than 2% is often cited). But perhaps more importantly, the system should *log no faster than traditional console logging on terabyte-quantity workloads* (100 KBps max). This means that it should *not* log all non-determinism, and in particular, all disk and network traffic. The ensuing logging rates, amounting to petabytes/week across all datacenter nodes, not only incur throughput losses, but also call for additional storage infrastructure (e.g., another petabyte-scale distributed file system).

Whole-Cluster Replay. The system should be able to replay-debug *all or any subset of nodes* used by the distributed application, if desired, after a failure is observed. Whole-cluster replay is essential because a failure or its underlying error may occur on any node in the system, and unfortunately, operators have no way of reliably anticipating the precise node(s) on which the error(s) will manifest.

Providing whole-cluster replay-debugging is challenging because datacenter nodes are often inaccessible at the time a user wants to initiate a replay session. Node failures, network partitions, and unforeseen maintenance are usually to blame , but without the recorded information on all those nodes, replay-debugging cannot be provided for any node. Traditional replication techniques may be employed to provide whole-cluster replay (e.g., by backing up logs to reliable storage), but the penalties in network bandwidth and disk storage overhead on petabyte workloads conflicts with the always-on operation requirement.

Wide Applicability. The system should make few assumptions about its environment. In particular, it should record and replay *arbitrary* user-level applications on modern commodity hardware *with no administrator or developer effort*. This means that it should not assume special hardware, languages, programming models (e.g., message passing or distributed shared memory), or access to source-code—the latter to accommodate the presence of blackbox devices and applications (e.g., proprietary routers or third-party libraries).

The commodity hardware requirement is essential because we want to replay existing datacenter systems as well as future systems Special languages and source-code modifications (e.g., custom APIs and annotations, as used in R2 [27]) are undesirable because they are cumbersome to learn, maintain, and retrofit onto existing datacenter applications. Source-

code analysis is also prohibitive due to the difficulty of analyzing large scale systems and the presence of blackbox components.

1.2 Contributions

To meet all of the aforementioned requirements, we've designed and implemented DCR—a Data Center Replay system that records and replays runs of datacenter applications like Cloudstore [1], Hypertable [2], and Memcached [4]. DCR meets its design requirements by leveraging the following key ideas.

Control-Plane Determinism. The key observation behind DCR is that, for debugging, we don't need a precise replica of the original production run. Instead, it often suffices to produce some run that exhibits the original run's *control-plane* behavior. The control-plane of a datacenter system is the code responsible for managing or controlling the flow of data through a distributed system. An example is the code for locating and placing blocks in a distributed file system.

The control plane tends to be complicated—it accounts for 99% of the code and bugs in datacenter applications—and thus serves as the breeding ground for bugs in datacenter software. But at the same time, the control-plane often operates at very low data-rates—it accounts for just 1% of all application traffic. Hence, by relaxing the determinism guarantees to control-plane determinism, DCR circumvents the need to record most inputs, and consequently achieves low record overheads with tolerable sacrifices of replay fidelity. We discuss control-plane determinism in detail in Chapter 3.

Deterministic-Run Inference. The central challenge in building DCR is that of reproducing the control-plane behavior of a datacenter application without knowledge of its original data-plane inputs. This is challenging because the control-plane's behavior depends on the data-plane's behavior. For example, a Hadoop Distributed Filesystem (HDFS) client's decision to look up a block in another HDFS data-node (a control plane behavior) depends on whether or not the block it received passed checksum verification (a data-plane behavior).

To address this challenge, DCR employs Deterministic-Run Inference (DRI)—a novel post-record inference procedure that computes unrecorded non-determinism (e.g., data-plane inputs, thread schedules) consistent with the recorded control-plane inputs and outputs (I/O) of the original run. Once computed, DCR then substitutes the resulting data-plane inputs and thread schedule, along with the recorded control-plane inputs, into subsequent program runs to generate a control-plane deterministic run. Chapter 5 discusses DRI and its operation in depth.

1.3 Outline and Summary

The remainder of this dissertation is organized as follows.

Chapter 2 provides background on datacenter applications and on the fundamentals of deterministic replay technology. We focus on why replay debugging capability is essential in the datacenter context, the challenges that must be overcome to provide it, and how our proposed replay system, DCR, differs from existing systems. The take-away point is that DCR's ability to efficiently record clusters of commodity multiprocessors running data-intensive applications sets it apart from other replay systems.

Chapter 3 presents the central hypothesis underlying DCR's design—that focusing on the control-plane of a datacenter application is the key to practical datacenter replay. The chapter introduces the notion of control plane determinism and uses experimental evidence to argue that control plane determinism indeed suffices for debugging datacenter applications. The take-away point is that control plane code accounts for most bugs (99%), yet is responsible for just a tiny fraction of I/O (1%), and therefore, a control plane deterministic replay system can, in practice, reproduce most bugs *and* incur low tracing overheads.

In Chapter 4, we provide an overview of DCR's design, focusing in particular on how DCR leverages the notion of control-plane determinism to meet its design requirements. In short, control-plane determinism enables DCR to record just control plane I/O. The low data rate nature of the control plane enables DCR to achieve low overhead recording for all cores in the cluster, thus precluding the need for specialized hardware or programming language support.

Chapter 5 details Deterministic-Run Inference (DRI)—a novel technique by which DCR achieves control-plane determinism despite not having recorded all sources of non-determinism in the original run. As discussed in depth, DRI leverages the power of program verification techniques and constraint solvers to compute, in an offline manner, the unrecorded nondeterminism. DRI is the heart of DCR, and ensuring control-plane determinism would be difficult without it.

Chaper 6 discusses key implementation details behind DCR, while Chapter 7 evaluates DCR, both in terms of its effectiveness as a debugging aid and its performance. To measure effectiveness, we present debugging case studies of real-world bugs (some new, some previously solved). We found that DCR was useful in reproducing and understanding them. To illustrate the tradeoff between recording overhead and inference time, we measured performance for two different configurations of DCR. We found one configuration to be useful for multi-processor intensive workloads and the other for I/O intensive workloads.

Finally, Chapter 8 concludes this dissertation with a discussion of DCR's key limitations. Though we have made much progress towards a datacenter replay system, much more work remains to be done. Toward that end, we present several ways in which one may improve DCR's practicality.

Chapter 2

Background

In this chapter, we develop context for DCR's design by providing background on datacenter applications (Section 2.1), deterministic replay technology (Section 2.2), and DCR's contributions in relation to existing work (Section 4.3).

2.1 Datacenter Applications

Datacenter applications are distributed programs that leverage the computing power of large clusters of commodity machines to process terabytes of data per day. These applications typically operate within the *datacenter*—large, air-conditioned buildings that house thousands of machines. Examples include the Bigtable distributed key-value store [12], the Hadoop MapReduce parallel data-processing framework [14], and the Google Distributed File System [26]. Datacenter applications are the workhorses of modern web services and applications. Google's web search service, for instance, uses these applications to perform indexing of the web (for fast searching) and tallying of their massive click logs (to charge advertising customers), while Facebook uses them to relay and organize updates from its millions of users. The remainder of this section describes the distinguishing characteristics of datacenter applications in greater detail and then explains why they are hard to debug.

2.1.1 Characteristics

Datacenter applications are marked by several characteristics. Foremost, they employ **large-scale data processing** on petabyte ("web-scale") quantity data. Google services, for example, generates terabytes of advertising click-logs, and all of it needs to be tallied to charge advertising customers. To operate on such large amounts of data, datacenter application employ a **scalable, distributed architecture**. The Hypertable distributed key-value store, for instance, employs a single master node to coordinate data placement, while multiple slave nodes are responsible for hosting and processing the data. Storage is typically scaled

through the use of a distributed file system. Examples include the Google Distributed File System (GFS) and Hadoop DFS (HDFS).

A hallmark of datacenter applications is that they run on top of **commodity hardware and software** platforms. In particular, modern datacenter design prefers off-the-shelf multiprocessor machines rather than specially designed, monolithic supercomputers. These commodity machines typically have an ample number of cores (8 is common these days), a significant amount of RAM (16GB is not uncommon), a terabyte-scale disks. On the software side, datacenter applications are written in a variety of general-purpose programming languages (e.g., it's not unusual for one application to include components written in C++, Python, and Java) and may contain multi-threaded code (e.g., using the `pthreads` library). Linux is the dominant datacenter OS platform.

Datacenter applications use both **shared memory and message passing** to communicate between CPU cores. Shared memory communication occurs when multiple cores on the same node use standard load and store instruction to read and write memory cells visible to multiple CPUs (e.g., via virtual memory hardware). For example, Hypertable slaves fork multiple threads, all of which access, in a synchronized fashion, an in-memory red-black-tree with regular load/store instructions. Message passing is typically used to transfer large chunks of data (e.g., 64MB blocks in a distributed filesystem) to other nodes, usually with the aid of an in-kernel networking software stack. The protocol of choice is almost always TCP/IP, which offers reliable and in-order delivery, and comes standard with Linux.

Finally, datacenter applications typically achieve **fault-tolerance via recomputation**. Datacenter nodes and software are generally unreliable, as nodes often die and software often fails due to bugs. When such unfortunate events occur, computation must be redone. Datacenter applications typically handle this by persistently storing their input sets (often in reliable distributed storage such as HDFS) and re-executing the computation on the stored data sets upon failure. As described in Chapter 4 `DCR` leverages this persistent-input property to achieve lower overheads.

2.1.2 Obstacles to Debugging

Datacenter applications are hard to debug for several reasons. One such reason is **non-deterministic execution**. In particular, modern applications, datacenter or otherwise, rarely execute the same way twice. Thus, when a datacenter application fails in production, non-determinism precludes the use of traditional cyclic debugging techniques (i.e., the process of repeatedly executing the program) to home-in on the root cause of the failure. The developer, upon re-executing the program, often finds that the failure doesn't occur anymore.

The are two major reasons why programs are non-deterministic. First, programs are typically dependent on inputs that vary from execution to execution, where by inputs we mean the data-sets to be processed as well as data from devices (e.g., random number generators, the network, etc.). In all but the simplest of programs, input non-determinism means that simply re-executing a program will not reproduce the bug.

The second major source of non-determinism is multiprocessor *data races*. By data race we mean a pair of *concurrent* and *conflicting* memory accesses to the same memory location. Two accesses are concurrent if their ordering depends on timing rather than principled synchronization. Two accesses are conflicting if at least one access is a write. Data races are often errors brought on by a failure to properly synchronize concurrent accesses to a shared data structure. But quite frequently, data races are benign and intentionally placed to obtain optimal performance. In either case, these sources of non-determinism often preclude the reproduction of bugs that occur later in the execution.

Another obstacle to debugging datacenter applications is **distributed state**. That is, unlike sequential programs, datacenter applications typically distribute their state among all nodes in the cluster. This in turn means that bugs in datacenter applications may span multiple nodes in the cluster. Debugging distributed state is hard because, to do so, one must first obtain a causally consistent view of it (also called a "distributed snapshot" [11]). In turn, obtaining such a snapshot is hard because distributed applications typically don't operate in synchrony with a global clock. Rather, they run on a cluster of machines each with loosely-synchronized clocks. Algorithms to obtain consistent snapshots of distributed application state do exist (e.g., via Chandy and Lamport [11]), but such techniques are usually too heavyweight to be effective for debugging.

Finally, a key challenge in debugging datacenter applications is of a non-technical nature: as **economies of scale** are driving the adoption of commodity clusters, the tolerable runtime overhead actually decreases—making up for a 50% throughput drop requires provisioning twice more machines; 50% of a 10,000-node cluster is a lot more expensive than 50% of a 100-node cluster. In this economy, extensive in-production instrumentation and online debugging (where one pauses execution while debugging and resumes it afterward) are prohibitive. In fact, when operating large clusters, it actually becomes cheaper to hire more engineers to debug problems than to buy more machines to tolerate the runtime overhead resulting from debugging instrumentation.

2.2 Deterministic Replay

Deterministic replay has been a reasonably active research subject for over two decades. Most of this work has centered on efficient logging for scientific computing applications, targeting especially multiprocessors and distributed shared memory computers; for an overview of the field we recommend an early survey by Dionne et al. [16] and later ones by Huselius [28] and Cornelis et al. [13]. Here we focus on background most relevant to upcoming, starting with guarantees offered by traditional replay systems, the challenges in providing those guarantees, and alternatives to deterministic replay.

int status = `ALIVE`, int *reaped = `NULL`

Master (Thread 1; CPU 1)	Worker (Thread 2; CPU 2)
1 r0 = status	1 r1 = input
2 if (r0 == DEAD)	2 if (r1 == DIE or END)
3 *reaped++	3 status = DEAD

Figure 2.1: Benign races can prevent even non-concurrency failures from being reproduced, as shown in this example adapted from the Apache web-server. The master thread periodically polls the worker's status, without acquiring any locks, to determine if it should be reaped. It crashes only if it finds that the worker is `DEAD`.

(a) Original	(b) Value-deterministic	(c) Non-deterministic
2.1 `r1 = DIE`	2.1 `r1 = DIE`	2.1 `r1 = DIE`
2.2 `if (DIE...)`	2.2 `if (DIE...)`	2.2 `if (DIE...)`
2.3 `status = DEAD`	1.1 `r0 = DEAD`	1.1 `r0 = ALIVE`
1.1 `r0 = DEAD`	2.3 `status = DEAD`	2.3 `status = DEAD`
1.2 `if (DEAD...)`	1.2 `if (DEAD...)`	1.2 `if (ALIVE...)`
1.3 `*reaped++`	1.3 `*reaped++`	
`Segmentation fault`	`Segmentation fault`	*no output*

Figure 2.2: The totally-ordered execution trace and output of (a) the original run and (b-c) various replay runs of the code in Figure 2.1. Each replay trace showcases a different determinism guarantee.

2.2.1 Determinism Models

The classic guarantee offered by traditional replay systems is *value determinism*. Value determinism stipulates that a replay run reads and writes the same values to and from memory, at the same execution points, as the original run. Figure 2.2(b) shows an example of a value-deterministic run of the code in Figure 2.1. The run is value-deterministic because it reads the value `DEAD` from variable `status` at execution point 1.1 and writes the value `DEAD` at 2.3, just like the original run.

Value determinism is not perfect: it does not guarantee causal ordering of instructions. For instance, in Figure 2.2(b), the master thread's read of `status` returns `DEAD` even though it happens before the worker thread writes `DEAD` to it. Despite this imperfection, value determinism has proven effective in debugging [10] for two reasons. First, it ensures that program output, and hence most operator-visible failures such as assertion failures, crashes, core dumps, and file corruption, are reproduced. Second, within each thread, it provides memory-access values consistent with the failure, hence helping developers to trace the chain of causality from the failure to its root cause.

2.2.2 Challenges

The key challenge of building a value-deterministic replay system is in reproducing **multiprocessor data-races**. Data-races are often benign and intentionally introduced to improve performance. Sometimes they are inadvertent and result in software failures. Regardless of whether data-races are benign or not, reproducing their values is critical. Data-race nondeterminism causes replay execution to diverge from the original, hence preventing downstream errors, concurrency-related or otherwise, from being reproduced. Figure 2.2(c) shows how a benign data-race can mask a null-pointer dereference bug in the code in Figure 2.1. There, the master thread does not dereference the null-pointer `reaped` during replay because it reads `status` before the worker writes it. Consequently, the execution does not crash like the original.

Several value-deterministic systems address the data-race divergence problem, but they fall short of our requirements. For instance, content-based systems record and replay the values of shared-memory accesses and, in the process, those of racing accesses [10]. They can be implemented entirely in software and can replay all output-failures, but incur high record-mode overheads (e.g., 5x slowdown [10]). Order-based replay systems record and replay the ordering of shared-memory accesses. They provide low record-overhead at the software-level, but only for programs with limited false sharing [17] or no data-races [44]. Finally, hardware-assisted systems can replay data-races at very low record-mode costs, but require non-commodity hardware [38, 39].

2.2.3 Alternatives

Given the challenges of deterministic replaying applications, particularly in the datacenter context, it is natural to wonder if alternative approaches are better suited to address the debugging problem. Here we provide a brief overview of alternatives to deterministic replay and argue that, though they have their merits, in the end they are complementary to deterministic replay technology.

Bug finding tools are popular debugging aids, especially because they are effective in finding bugs in programs before they are deployed. Such tools employ static analysis, verification, testing, and model-checking [18, 33, 40], and are capable of finding hundreds of bugs in real code bases. However, they aren't perfect—they may miss bugs, especially in large state-space applications written in unsafe programming language, which includes many datacenter applications. These bugs eventually filter through to production, and when a failure results, a deterministic-replay system would still be useful in reproducing those failures.

Another alternative to record-replay is **deterministic execution** (a.k.a., deterministic multi-threading). Like record-replay, deterministic execution provides a deterministic replay of applications. But unlike record-replay, its primary goal is to minimize the amount of non-determinism naturally exhibited by application, so that they behave identically, or at least similarly, across executions without having to record much [9,15,42]. Deterministic execution is complementary to record-replay because the less non-determinism there is, the less a record-replay system must record. However, deterministic execution cannot eliminate non-determinism entirely, since environmental non-determinism (i.e., non-determinism coming from untraced entities) still has to be recorded. Thus, we argue that some form of record-replay is still needed in the end.

Developers often employ **log-based debugging** techniques in the absence of deterministic replay. These techniques analyze application-produced log files to help with debugging. The SherLog system, for instance, pieces together information about program execution given just the program's console log [50], while automated console log analysis techniques [49] automatically detect potential bugs using machine learning techniques. Unlike record-replay, these techniques impose no additional in-production runtime overhead. However, the inference these systems draw are fundamentally limited by developer-instrumented console logs: for example, they are unable to reconstruct detailed distributed execution state, which is necessary for deep inspection of distributed application behavior.

Perhaps the most widely used approach to debugging is **online debugging and checking**, whereby error checking is performed while the application is running in production. The simplest form of this is the classic inline assertion check, but more sophisticated techniques are possible, such as attaching to a production run with GDB to print a stack trace, or using "bug-fingerprints" [51] to check for deadlocks and other common errors. Online debugging is often more lightweight than deterministic replay. After all, the programmer gets to decide precisely what information is checked. On the other hand, it often requires foresight or a

hunch as to the underlying root cause and, in the face of non-determinism, does not permit cyclic debugging. In contrast, deterministic replay requires no foresight (as it reproduces the entire execution rather than select portions of it) and may be used for cyclic debugging.

Chapter 3

The Central Hypothesis

The central hypothesis underlying DCR's design is that, for debugging datacenter applications, we do *not* need a precise replica of the original production run. Rather, it often suffices to produce some run that exhibits the original run's *control-plane* behavior. In other words, we hypothesize that *control-plane determinism* is sufficient for debugging datacenter applications. We begin this chapter by defining the concept of control-plane determinism. Then we present a method by which we test the sufficiency of control-plane determinism. Using this test, we then experimentally verify that control-plane determinism does indeed suffices for real-world applications.

3.1 Control-Plane Determinism

We say that a replay run of a program exhibits *control-plane determinism* (i.e., is control-plane deterministic) if its control-plane code obtains the same inputs and produces the same outputs as in the original program run. The key to understanding this definition is in understanding what we mean by control-plane code. In the remainder of this section, we define control plane code and then explain how control plane determinism differs from traditional models of determinism.

3.1.1 The Control and Data Planes

The *control plane* of a datacenter application is the code that manages or controls user-data flow through the distributed system. Examples of control-plane operations include locating a particular block in a distributed file-system, maintaining replica consistency in a meta-data server, or updating routing table entries in a software router. The control plane is widely thought to be the most bug-prone component of datacenter systems. But at the same time, it is thought to consume only a tiny fraction of total application I/O.

Figure 3.1(a) gives a concrete example of control plane code, adapted from the CloudStore

(a) Control Plane Code

```
void handle_new_block(struct block *b) {
  if (crc32(b->buf, b->len) != b->crc) {
    printf("checksum_mismatch\n");
    recover_from_backups(b->id);
  }
}
```

(b) Data Plane Code

```
int crc32(void *buf, size_t len) {
  int crc = 0;
  for (i = 0; i < len; i++) {
    crc = crc32tab[(crc ^ buf[i]) & 0xff];
  }
  return crc;
}
```

Figure 3.1: Control and data-plane code for checking the integrity of user data blocks, adapted from the CloudStore distributed file system. (a) Control plane code performs administrative tasks such as detecting and recovering corrupted file-system blocks. (b) In contrast, data plane code performs data processing tasks such as computing block checksums.

distributed file-system. The code checks the integrity of file-system data blocks and if they are corrupt, it performs a recovery action. It exhibits the hallmarks of control plane code. In particular, the code is administrative in nature: it simply checks if data is properly flowing through the system via checksum comparison. Moreover, it does no direct data-processing on user data blocks, but rather invokes specialized functions (crc32) to do the data processing for it. Finally, the code is complex in that it invokes distributed algorithms (e.g., code that maintains a mapping from blocks to nodes) to recovers corrupted blocks.

Unlike control-plane code, *data-plane* code is the workhorse of a datacenter application. It is the code that processes user data, often byte by byte, on behalf of the control plane, and its results are often used by the control plane to make administrative decisions. Data-plane tends to be simple—a requirement for efficiently processing large amounts of user data—and is often part of well-tested libraries. Examples include code that computes the checksum of an HDFS file-system block or code that searches for a string as part of a MapReduce job. The data plane is widely thought to be the least bug-prone component of a datacenter system. At the same time, experience indicates that it is responsible for a majority of datacenter traffic.

Figure 3.1(b) gives a concrete example of data plane code, also adapted from the CloudStore distributed file-system. The code simply computes the checksum (CRC32) of a given file-system data block using a crc lookup table (`crc32tab`), and exhibits several hallmarks of data-plane code. For instance, this particular crc32 routine is well-tested and simple: it came from a well-tested library function, and was implemented in under 10 lines of code (excluding the lookup table). The code is also high data rate: crc32 is a CPU-bound task that requires inspecting data blocks byte-by-byte. Finally, the code is invoked by control plane code and its results are used to make administrative decisions.

3.1.2 Comparison With Value Determinism

Control-plane determinism dictates that the replay run exhibits the same control-plane I/O behavior as the original run. For example, control-plane determinism guarantees that the application given in Figure 3.1 will output the same console-log error message ("checksum mismatch") as the original run.

Control-plane determinism is weaker than traditional determinism models such as value determinism (see Section 2.2): it makes no guarantees about non-control plane I/O properties of the original run. For instance, control-plane determinism does not guarantee that the replay run will read and write the same data-plane values as the original. This means that the contents of the file-system block structure (e.g., block id, block data, etc.) may be different in a control-plane deterministic replay run. Moreover, because the data may be different, control-plane determinism does not guarantee that the replay run will take the same program path (i.e., sequence of branches) as the original run.

Despite the relaxation, we argue that control-plane determinism is effective for debugging purposes, for two reasons. First, control-plane determinism ensures that control-plane output-visible failures, such as console-log error messages, assertion failures, crashes, core dumps, and file corruption, are reproduced. A control-plane deterministic run will, for instance, output "checksum mismatch" just like in the original run. Second, control-plane deterministic runs provide memory-access values that, although may differ from the original values, are nonetheless consistent with the original control-plane visible failure. For instance, even though a block data contents and hence its computed checksum may differ from the original run, they are guaranteed to be consistent with one other.

The chief benefit of control-plane determinism over value determinism is that it does not require the values of data-plane I/O and data races to be the same as the original values. In fact, by shifting the focus of determinism to control-plane I/O rather than values, control-plane determinism enables us to circumvent the need to record and replay data-plane I/O and data-races altogether. Without the need to reproduce data-plane I/O and data-race values, we are freed from the tradeoffs that encumber traditional replay systems. The result, as we detail in Chapter 4, is `DCR`—a Data Center Replay system that meets all of our requirements.

3.2 Testing the Hypothesis

We present the criteria for verifying our hypothesis that control-plane determinism suffices, and then describe the central challenge in its verification.

3.2.1 Criteria and Implications

To show that our hypothesis holds, we must empirically demonstrate two widely held but previously unproven assumptions about the control and data planes.

Bug Rates. First, we must show that the control plane rather than the data plane is *by far* the most bug prone component of datacenter systems. If the control plane is the most bug prone, then a control-plane deterministic replay system will have high replay fidelity–it will be able to reproduce most application bugs. If not, then control plane determinism will have limited use in the datacenter, and our hypothesis will be falsified.

Data Rates. Second, we must show that the control plane rather than the data plane is *by far* the least data intensive component of datacenter systems. If so, then a control plane deterministic replay system is likely to incur negligible record mode overheads – after all, such a system need not record data plane traffic. If, however, the control plane has high data rates, then it is likely to be too expensive for the datacenter, and our hypothesis will be falsified.

3.2.2 The Challenge: Classification

To verify our hypothesis, we must first classify program instructions as control or data plane instructions. Achieving a perfect classification, however, is challenging because the notions of control and data planes are tied to program semantics, and thus call for considerable developer effort and insight to distinguish between them. Consequently, any attempt to manually classify every instruction in large and complex applications is likely to provide unreliable results.

To obtain a reliable classification with minimal manual effort, we employ a semi-automated classification method. This method operates in two phases. In the first phase, we manually identify *user data* flowing into the distributed application of interest. By user data we mean any data inputted to the distributed application with semantics clear to the user but opaque to the system (e.g., a file to be uploaded into a distributed file-system). We identify user data by the files in which it resides.

In the second phase, we automatically identify the static program instructions influenced by the previously identified user data. For this purpose, we employ a *whole distributed system taint-flow analysis*. This distributed analysis tracks user data as it propagates through nodes in the distributed system. Any instructions tainted by user data are classified as data plane instructions; the remaining untainted but executed instructions are classified as control plane instructions.

In the remainder of this section we first present the details of our distributed taint-flow analysis and then we discuss the potential flaws of our classification method.

Tracking User Data Flow

To track user data through the distributed application, we employ an instruction-level (x86), dynamic, and distributed taint flow analysis. We choose an instruction level analysis because datacenter applications are often written in a mix of languages. We choose a dynamic analy-

sis because datacenter applications often dynamically generate code, which is hard to analyze statically. Finally, we seek a distributed analysis because we want to avoid the error-prone task of manually identifying and annotating user-data entry points for each component in the distributed system.

Propagating Taint. Unlike single-process taint-flow analyses such as TaintCheck [41], our analysis must track taint both within a node (e.g., through loads and stores) and across nodes (e.g, through network messages).

Within a Node. We propagate taint at byte granularity largely in accordance with the taint-flow rules used by other single-node taint-flow analyses [45]. For instance, we taint the destination of an n-ary operation if and only if at least one operand is tainted. Our analysis does, however, differ from others in two key details. First, we create new taint only when bytes are read from designated user data files (as opposed to all input files or network inputs). And second, we do *not* taint the targets of tainted-pointer dereferences unless the source itself is tainted (this avoids misclassifying control plane code, see Section 3.2.2).

Across Nodes. To propagate taint across nodes, we piggyback taint meta-data on tainted outgoing messages. We represent taint meta-data as a string of bits, where each bit indicates whether or not the corresponding byte in the outgoing message payload is influenced by user data. The receiving process extracts the piggybacked taint meta-data and applies taint to the corresponding bytes in the target application's memory buffer.

We piggyback meta-data on outgoing UDP and TCP messages with the aid of a transparent message tagging protocol we developed in prior work [25]. For UDP messages, the protocol prefixes each outgoing UDP message with meta-data and removes it upon reception. For TCP messages, the protocol inserts meta-data into the stream at `sys_send()` message boundaries, along with the size of the message. On the receiving end, the protocol uses the previous message's size to locate the meta-data for the next message in the stream.

Reducing Perturbation. A key difficulty in performing taint-analysis on a running system is that the high overhead of analysis instrumentation (approximately 60x in our case) severely alters system execution. For instance, in our experiments with OpenSSH [5], taint-flow instrumentation extended computation time so much that `ssh` consistently timed out before connecting to a remote server. This precluded any analysis of the server.

To reduce perturbation, we leverage (ironically) deterministic replay technology. In particular, we perform our taint-flow analysis offline on a deterministically replayed execution rather than the original execution. The key observation behind this approach is that collecting an online trace for deterministic replay is much cheaper than performing an online taint-flow analysis (a slowdown of 1.8x vs. 60x). Hence, by shifting the taint-analysis to the replay phase, we eliminate most unwanted instrumentation side-effects.

To obtain a replay execution suitable for offline taint-flow analysis, we employ the Friday

distributed replay and analysis platform [24]. Friday records a distributed system's execution and replays it back in causal order (i.e., respecting the original ordering of sends and receives). Friday was not designed for datacenter operation—it records both control and data plane inputs and hence is too expensive to deploy in production. Nevertheless, it is sufficient for the purposes of collecting and analyzing production-like runs.

Accuracy

Though we believe our method to be more reliable than manual classification, it has limitations that may reduce its precision. We first describe these limitations and then discuss their impact on our classification results.

Sources of Imprecision. There are two key sources of imprecision.

User Data Misidentification. It is possible that we may fail to identify user data files. As a result, some data plane code will be erroneously classified as control plane code. We may also mistakenly designate non-user data files as user-data files. In that case, control plane code will be misclassified as data plane code. Despite these dangers, we note that the possibility of misidentification is very low in practice: our evaluation workloads are composed of only a few user data files that we hand picked (see Section 3.3.1).

Tainted Pointers. Our policy of not tainting the targets of tainted-pointer dereferences (unless the source itself is tainted) may result in data plane code being misclassified as control plane code. An example is the following snippet from a C implementation of CRC32 used in OpenSSH [5]:

```
...
crc = crc32tab[(crc ^ buf[i]) & 0xff];
...
```

Our pointer-insensitive analysis will not taint the value of `crc` as it should, for the following reason. Rather than compute the CRC mathematically, the code looks up a pre-computed table of constants (`crc32tab`). Even though the table index (`buf[i]`) is tainted, the value in the corresponding table entry is a constant, and thus our analysis will assume that `crc` is untainted as well.

Despite its drawback, we chose a pointer-insensitive analysis because it avoids the large number of data plane misclassifications produced by a pointer-sensitive analysis. An example of such misclassification can be seen in the following C code snippet:

```
int h = hash(user_data);
pthread_mutex_lock(&array[h].lock);
```

Both pointer sensitive and insensitive policies will correctly classify the hash computation as a data-plane operation. However, a pointer-sensitive policy will also classify the lock

acquisition (a control plane operation) as a data plane operation. Reads of the lock variable must be dereferenced by the tainted hash code, after all. Unfortunately, such code is common in some applications we've worked with (e.g., Hypertable [2]).

To compensate for the under-tainting resulting from our pointer-insensitive policy, we manually identify the data plane code that is missed. We perform this manual identification with the aid of a pointer-sensitive version of our analysis. Specifically, we comb the results of the pointer-sensitive analysis, to the best of our ability, for data plane code that would have been missed with an insensitive policy. We identified the CRC32 example given above in this manner, for instance. In the future, we hope to automate the weeding-out process in order to reduce human error.

Impact on Results. Overall, the above imprecisions in our method are more likely to induce under-tainting rather than over-tainting. In other words, we are more likely to misclassify data plane code as control plane code. Such misclassification will produce unsound bug rate results. In particular, if we observe a high control plane bug rate, then all of those bugs may not stem from control plane code–some, perhaps a sizeable portion, may stem from data plane code. By contrast, the data rate results will remain sound despite under-tainting. Specifically, if we observe a high data plane rate (as we indeed do, see Section 3.3.3), then those results are accurate. After all, under-tainting can only decrease the measured data plane rate.

Completeness

The results produced by our classifier do not generalize to arbitrary program executions. The reason is that our taint-flow analysis is dynamic rather than static, and therefore we have no way to classify instructions that do not execute in a given run. Though we cannot completely overcome this limitation, we compensate for it by performing our taint-flow analysis on multiple executions with a varied set of inputs (see Section 3.3.1) We ultimately classify only those instructions executed in at least one of those runs. In future work, we hope to increase the quantity and quality of inputs to derive a more general result.

3.3 Verifying the Hypothesis

We evaluate our hypothesis on real datacenter applications per the criteria given in Section 3.2.1. In short, we found that both clauses of our testing criteria held true. That is, we found that control plane code is the most complex and bug prone (with an average per-execution code coverage and reported bug rate of 99%), and that data plane code is the most data intensive (accounting for an average 99% of all application I/O). Taken together, these results suggest that, by relaxing determinism guarantees to control-plane determinism, a replay system will be able to provide both low-overhead recording and high fidelity replay.

3.3.1 Setup

Applications. We test our hypothesis on three real-world datacenter applications: *CloudStore* [1], *Hyptertable* [2], and *OpenSSH* [5].

CloudStore is a distributed filesystem written in 40K lines of multithreaded C/C++ code. It consists of three sub-programs: the master server, slave server, and the client. The master program maintains a mapping from files to locations and responds to file lookup requests from clients. The slaves and clients store and serve the contents of the files to and from clients.

Hypertable is a distributed database written in 40K lines of multithreaded C/C++ code. It consists of four key sub-programs: the master server, meta-data server, slave server, and client. The master and meta-data servers coordinate the placement and distribution of database tables. The slaves store and serve the contents of tables placed there by clients.

OpenSSH is a secure communications package widely used for securely logging in (via `ssh`) and transferring files (via `scp`) to and from remote nodes. In addition to these client side components, OpenSSH requires the use of a server (`sshd`) on the target host, and optionally, a local authentication agent (`ssh-agent`) responsible for storing the client's private keys. The package consists of 50K lines of C code.

Workloads. We chose large user data files to approximate datacenter-scale workloads. Specifically, for *Hypertable*, 2 clients performed concurrent lookups and deletions to a 10 GB table of web data. Hypertable was configured to use 1 master server, 1 meta-data server, and 1 slave server. For *CloudStore*, we made one client put a 10 GB gigabyte file into the filesystem. We used 1 master server and 1 slave server. For *OpenSSH*, we used `scp` (which leverages `ssh`) to transfer a 10 GB file from a client node to a server node running `sshd`. We conducted 5 trials, each with a different input file and varying degrees of CPU, disk, and network load.

3.3.2 Bug Rates

Metrics. We gauge bug rates with two metrics: *plane code size* and *plane bug count*. Plane code size is the number of static instructions in the control or data plane of an application, as identified by our classifier (see Section 3.2.2). Code size is a good approximation of code bug rate since it indirectly measures the code's complexity and thus its potential for defects. Plane bug count is the number of bug reports encountered in each component over the system's development lifetime, and serves as direct evidence of a plane's bug rate.

We measured plane code size by looking at the results of our classification analysis (see Section 3.2.2) and counting the number of static instructions executed by each plane across all test inputs. We measured the plane bug count by inspecting and understanding, *at the high level*, all reported, non-trivial defects in the application's bug report database. For each defect, we isolated the relevant code and then used our understanding of the report and our

	Code Complexity (# of Insns.)		
Application	Control (%)	Data (%)	Total (K)
CloudStore			
Master	100	0	85
Slave	99.7	0.3	92
Client	99.7	0.3	55
Hypertable			
Master	100	0	95
Metadata	100	0	68
Slave	96.4	3.6	124
Client	99.7	0.3	143
OpenSSH			
Server	97.8	2.2	103
AuthAgent	100	0	11
Client	98.9	1.1	69
Average	99.2	0.8	85

Figure 3.2: Plane code complexity as a percentage of the number of static x86 instructions that were executed at least once in our runs. As hypothesized, the control plane accounts for almost all of the code in a datacenter application.

code classification to determine if it was a control or data plane issue.

Code Size Results. Figure 3.2 gives the measured size in static instructions for the control and data planes. At the high level, it shows that almost all of an application's code–99% on average–is in the control plane. Components such as the Hypertable Master and Metadata servers are entirely control plane. This is not surprising because these components don't access any user data; their role, after all, is to direct the placement of user data kept by the Range server. More interestingly, however, components that do deal with user data (e.g., the Hypertable Range server) are still largely control plane.

To understand why the control plane dominates even in the data intensive application components, we counted the number of distinct functions invoked by each plane. The results, shown in Figure 3.3, reveal that control plane code invokes many functionally distinct operations. For instance, we found that CloudStore's control plane must allocate and deallocate memory (calls for `malloc()` and `free()`), perform lookups on the directory tree (`Key::compare()`) to determine data placement, and prepare outgoing messages (`TcpSocket::Send()`), just to name a few. By contrast, Figure 3.3 shows that the data plane has extremely low function complexity: one function, in most cases, does almost all of the data plane work. To give an example, we found that almost all of the CloudStore

	Code Complexity (# of Functions)	
Application	Control	Data
CloudStore		
Master	261	0
Slave	93	1
Client	66	1
Hypertable		
Master	275	0
Metadata	208	0
Slave	464	74
Client	163	6
OpenSSH		
Server	100	1
AuthAgent	13	0
Client	27	1
Average	167	8

Figure 3.3: Plane code complexity as measured by the number of C/C++ functions hosting the top 90% of the most executed instruction locations in a plane. The control plane is more complex in that draws upon a vast array of distinct functions to carry out its core tasks, while the data plane relies on just a handful.

	Reported Bugs		
Application	Control (%)	Data (%)	Total
CloudStore			
Master	N/A	N/A	N/A
Slave	N/A	N/A	N/A
Client	N/A	N/A	N/A
Hypertable			
Master	100	0	5
Metadata	100	0	3
Slave	100	0	37
Client	93	7	14
OpenSSH			
Server	100	0	215
AuthAgent	100	0	2
Client	99	1	153
Average	98.8	1.2	72

Figure 3.4: Plane bug count. The control plane accounts for almost all reported bugs in an application. CloudStore numbers are not given because it does not appear to have a bug report database.

Client's data plane activity consists of calls to `adler32()` – a data checksumming function.

Bug Count Results. Figure 3.4 gives the number of bug reports for each plane. At the high level, it shows that an average 99% of bug reports stem from control plane errors. We were able to identify two reasons for this result.

The first reason is that significant portions of control plane code is new and written specifically for the unique and novel needs of the application. By contrast, the data plane code generally relies *almost exclusively* on previously developed and well-tested code bases (e.g., libraries). To substantiate this, we measured the percentage of instructions executed from within libraries and inlined C++/STL code by each plane. The results, given in Figure 3.5, show that a median 99.8% of instructions executed by the data plane come from well-tested libraries such as `libc` and `libcrypto`, while only a median 93% of instructions executed by the control plane come from libraries.

A second reason for the high control plane bug count is complexity. That is, the control plane tends to be more complicated. This is evidenced not only by the function complexity results in Figure 3.3, but also by the nature of the bugs themselves. In particular, our inspection of the source code revealed that control plane bugs tend to be more complex than data plane bugs–an artifact, perhaps, of the need to efficiently control the flow of large amounts of data. For instance, Hypertable migrates portions of the database from range

	Instructions Executed (Billions)			
	Control Plane		Data Plane	
Application	Lib (%)	Total	Lib (%)	Total
CloudStore				
Master	91.0	0.1	0	0
Slave	96.3	0.1	99.9	62
Client	94.4	0.1	99.9	55
Hypertable				
Master	93.3	1	0	0
Metadata	92.8	1	0	0
Slave	90.3	1	88.3	2732
Client	89.8	1	98.2	3158
OpenSSH				
Server	93.6	0.8	99.6	1280
AuthAgent	82.7	0.02	0	0
Client	96.2	0.9	100	1301

Figure 3.5: The percentage of dynamic x86 instructions issued from well-tested libraries (e.g., code found in `libc, libstdc++, libz, libcrypto`, etc.) and inlined template code (from C++ header files), broken down by control and data planes. The data plane relies almost exclusively on well-tested code, while the control plane contains sizable portions of custom code.

	I/O Traffic		
Application	Control (%)	Data (%)	Total (GB)
CloudStore			
Master	100	0	0.2
Slave	1.7	98.3	20.4
Client	1.6	98.4	20.4
Hypertable			
Master	100	0	0.2
Metadata	100	0	0.3
Slave	1.4	98.6	20.5
Client	1.5	98.5	20.6
OpenSSH			
Server	0.8	99.2	20.2
AuthAgent	100	0	0.001
Client	0.6	99.4	20.2

Figure 3.6: Input/output (I/O) traffic size in gigabytes broken down by control and data planes. For application components with high data rates, almost all I/O is generated and consumed by the data plane.

server to range server in order to achieve an even data distribution. But the need to do so introduced Hypertable issue 63 [3]—a data corruption bug that triggers when clients concurrently access a migrating table.

3.3.3 Data Rates

Metric. We measure the number of input/output (I/O) bytes transferred by each plane. Data is considered input if the plane reads the data from a communication channel, and output if the plane writes the data to a communication channel. By communication channel, we mean a file descriptor that connects to the tty, a file, a socket, or a device. To measure the amount of I/O, we interposed on common inter-node communication channels via system call interception. If the data being read/written was tainted by user data, then we considered it data plane I/O plane; otherwise it was treated as control plane I/O.

Results. Figure 3.6 gives the data rates for the control and data planes. At the high level, the results show that the control plane is by far the least data intensive component. Specifically, the control plane code accounts for an average 1% of total application I/O in components that have a mix of control and data plane code (e.g., Hypertable Slave and Client). Moreover, in components that are exclusively control plane (e.g., the Hypertable Master), the overall I/O rate is orders of magnitude smaller than those that have data plane

code. These results highlight a key benefit of a control plane deterministic replay system: it provides a drastic reduction in logging overhead that in turn enables low-overhead, always-on recording.

Chapter 4

System Design

We begin this chapter with an overview of the approach behind DCR's design (Section 4.1). Then we detail how we translate this approach into a concrete system architecture (Section 4.2). We finish by comparing DCR's design with those of prior replay systems (Section 4.3).

4.1 Approach

Two ideas underly DCR's design. The first is the idea of *relaxing determinism guarantees*. Specifically, DCR aims for *control-plane determinism*—a guarantee that replay runs will exhibit identical control-plane I/O behavior to that of the original run. Control-plane determinism is important because, unlike stronger determinism models, it circumvents the need to record anything but control-plane I/O. In particular, data-plane inputs (which have high data-rates—see Section 3) and thread schedules (expensive to trace on multiprocessors) need not be recorded, thereby allowing DCR to efficiently record all nodes in the system, and without specialized hardware or programming languages. Chapter 3 gives further background on control-plane determinism.

Control-plane I/O alone is insufficient for a deterministic replay—other sources of non-determinism such as data-plane inputs and thread-schedules are required as well. Thus the second idea behind DCR is that of inferring unrecorded non-determinism (e.g., data-plane inputs, thread-schedules) in an offline, post-record phase. DCR performs this inference using a novel technique we call DRI. DRI uses well-known program verification techniques like verification condition generation [22] and constraint solving to compute the unrecorded non-deterministic data that is consistent with the recored control plane I/O. Once computed, DCR then substitutes the resulting non-deterministic data, along with the recorded control-plane inputs, into subsequent program runs to generate a control-plane deterministic run. Chapter 5 gives the details of DRI.

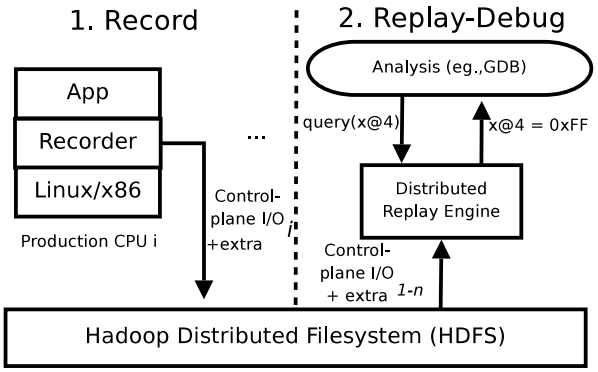

Figure 4.1: DCR's distributed architecture and operation. DCR's Distributed Replay Engine (DRE) uses the recorded control-plane I/O as well as any optionally recorded information (denoted by *extra*) to provide a control-plane deterministic replay. Additionaly recorded information speeds the process of inferring a replay run.

4.1.1 The Challenge

The key challenge in designing DCR is finding the right tradeoff point between recording overhead and inference time. At one extreme, control-plane determinism in conjunction with DRI enable us to record just control-plane I/O. However, the resulting inference time is impractically long. Details are presented in Chapter 5, but in brief, the reason is that DRI must explore the exponential-sized space of all data-plane inputs and thread-schedules. At the other extreme, DCR could record everything, including data-plane inputs and thread schedule. Inference would then not be required at all, but the recording overhead would be too high for prolonged production use—a key design requirement.

To reconcile the tension between recording overhead and inference time, we observe that any fixed point is likely to be inadequate for all applications, environments, and user needs. Therefore, DCR allows its users to specify what tradeoff point is best for their needs—a notion we term *user-selective recording and replay*.

User-selectivity is beneficial in many circumstances. For example, in many datacenter environments, data plane inputs are persistently stored. In such cases, the user may opt to record a pointer to these data plane inputs (in addition to control plane I/O) and reuse them at replay time, thus cutting inference costs considerably (why infer data-plane inputs if you have them already?). The cost of recording a pointer to data plane inputs is substantially lower than recording a copy of the inputs. To given another example, some I/O intensive applications are lite on sharing, so recording a partial ordering of memory accesses (e.g., with a page-based memory sharing protocol [17]) may result in small overheads. If the user is willing to tolerate this overhead (perhaps because he expects the recording to be done on only a portion of the datacenter cluster), then user-selectivity enables DCR to cut inference

time with this additional information.

4.2 Architecture

Figure 4.1 shows the user-selective recording and replay architecture of our control-plane deterministic replay-debugging system. A key feature is that DCR enables its user to choose what information is recorded, thereby enabling her to tradeoff recording overhead and inference time based on the application, its environment, and her needs. More specifically, DCR operates in two phases:

Record Mode. DCR records, *at the minimum*, *control-plane inputs and outputs* (I/O) for all production CPUs (and hence nodes) in the distributed system. Control-plane I/O refers to any inter-CPU communication performed by control-plane code. This communication may be between CPUs on different nodes (e.g., via sockets) or between CPUs on the same node (e.g., via shared memory).

DCR may also record, *at the user's option*, additional information about the original run. This information may include the path taken by each CPU in the run, a partial ordering of instruction interleaving (e.g., lock ordering), and/or a reference to the contents of persistent data-plane inputs (e.g., click-logs).

DCR streams all recorded information to a Hadoop Filesystem (HDFS) cluster—a highly available distributed data-store designed for datacenter operation.

Replay-Debug Mode. To replay-debug her application, an operator or developer interfaces with DCR's Distributed-Replay Engine (DRE). The DRE leverages the previously recorded trace data to provide the operator with a causally-consistent, control-plane deterministic view of the original distributed execution. The operator interfaces with the DRE using an analysis plug-in. For instance, the DCR offers a distributed data-flow plug-in that tracks the flow of data through the distributed execution (more details of analysis plug-ins are given in Section 4.2.3). The plug-in issues queries to DCR's Distributed-Replay Engine (DRE) that in turn leverages the previously recorded control-plane I/O and any additionally recorded information to provide the analysis plug-in with a causally-consistent, control-plane deterministic view of the original distributed execution.

4.2.1 User-Selective Recording

While control-plane I/O must be recorded, DCR optionally records additional data to speed the inference process. Here we describe in further detail what recording options are available, and how this information is traced.

Persistent Data-Plane Inputs

A key observation made by DCR's selective recorder is that, in many datacenter applications, data-plane input files (e.g., click logs) are persistently stored, typically on distributed storage (DFS). This assumption holds because datacenter applications keep their data-plane inputs around anyway for fault-tolerance purposes (i.e., to recompute if a machine goes down, or if there was a bug in the computation). DCR takes advantage of this property by recording a pointer to the data-plane inputs rather than a copy of it, thus providing a significant savings in runtime slowdown incurred and storage consumed.

The strategy of recording a pointer to the data rather than the data itself raises two questions. First, which data files should be considered persistent? And second, how can we tell that a particular program input comes from the persistent data files? Perhaps the simplest method, and the method adopted by DCR, is to have the user designate persistent storage via annotations. For example, DCR accepts annotations in the form of a list of URLs, where each URL has the form `hdfs://cluster12/`. Each URL is interpreted as a filesystem and path in which persistent files reside.

To determine if a particular program input acquired during execution comes from persistently stored files (and thus whether or not it should be recorded), DCR requires that the designated files come from a VFS-mounted filesystem (EXT3 and HDFS, for instance, has VFS-mount support). Then detecting whether data being read comes from persistent storage is simply a matter of intercepting system calls (e.g., `sys_read()`) and inspecting the system call arguments to determine if the associated file-descripted connects to a persistent file. If VFS-mounting is not supported by the filesystem (the uncommon case), then hooks into the filesystem code are necessary.

Paths

We define a program path as a sequence of branch outcomes on a given CPU. For conditional branches, the outcome is simply a boolean: true if the branch was taken and false otherwise. For indirect branches, the branch outcome is the target of the indirect jump. DCR enables the user to record the path taken by all CPUs or a subset of them.

DCR supports several ways to record the program path, each of which has its tradeoffs. Perhaps the most straightforward way is to use the hardware branch tracing support found in modern commodity machines (e.g., the Branch Trace Store facility in Intel Pentium IV and higher [29]). The benefit of hardware branch tracing is that it just works: no special instrumentation or assumptions about the software being traced need be made. The chief drawback is that it's slow: each branch induces a 128-bit write for each branch outcome into a small in-memory buffer that must be frequently flushed.

An alternative method is to employ software-based branch tracing, where we instrument each branch instruction and record its outcome. A naive logging approach would result in high memory bandwidth and disk I/O consumption, so DCR employs an on-the-fly compres-

sion technique to dynamically reduce the size of the branch trace data. The key observation behind this technique is that it suffices to log just the mispredicted branches. Assuming the branch predictor is deterministic, all predicted branches will be re-predicted precisely in subsequent runs. The key challenge with this approach is that the hardware predictor cannot be relied upon to be deterministic. Thus, rather than employ the hardware branch predictor, we perform branch tracing with an idealized software branch predictor (a standard 2-level predictor with a Branch Trace Buffer and a Return Stack Buffer).

Partial-Order of Instructions

DCR supports tracing two types of instruction partial orders: conflict order and lock order. Conflict order refers to the interleaving of conflicting memory accesses. We say that two accesses conflict iff they reference the same memory location and at least one of them is a write. Note that a trace of conflict ordering implies a trace of race ordering, since races are conflicting accesses that occur in parallel. DCR also supports tracing the lock ordering, which in many cases is more lightweight than tracing conflict ordering. By lock ordering, we mean the ordering of operations used for synchronization. Details of how each of these orders is traced follow.

Conflict-Order. A simple approach to tracing conflicting accesses is dynamically and at byte granularity. This approach is difficult, however, because it requires instruction-level instrumentation and tracing of memory access instructions. One way to instrument accesses is with binary translation techniques, but that incurs huge slowdowns. Another instrumentation method is to interpose on hardware coherence messages at the processor hardware level, but that requires specialized hardware or impractical changes to existing commodity architectures. Yet another approach is to employ static analysis to trace those accesses that may conflict (via a conservative analysis). But this assumes access to source code, which is not always possible in the datacenter environment.

Rather than trace conflicts at byte granularity, DCR traces it at page-granularity. Specifically, DCR employs the page-based Concurrent-Read Exclusive-Write (CREW) memory sharing protocol, first suggested in the context of deterministic replay by Instant Replay [34] and later implemented and refined by SMP-ReVirt [17]. Page-based CREW leverages page-protection hardware found in modern MMUs to detect concurrent and conflicting accesses to shared pages. When a shared page comes into conflict, CREW then forces the conflicting CPUs to access the page one at a time, and then records the ordering of accesses. Details of our CREW implementation are given in Section 6.2.1.

Lock-Order. DCR supports two ways to trace the lock order, and both have their trade-offs. The straightforward way is to instrument the threading library's (e.g., `libpthreads`) synchronization routines (e.g., `pthread_mutex_lock`, etc.). This requires access to the lock library source code or a willingness on the user's part to use a lock library provided with the

DCR distribution. Neither of these may be appealing in software stacks that must conform to rigid version requirements (e.g., typically done to ward off bugs due to unforeseen or incompatible library interactions). Moreover, such manual instrumentation may miss custom synchronization routines.

DCR also enables lock tracing via dynamic binary instrumentation. The key idea is to instrument just locked instructions (e.g., those with the lock prefix in x86). This may be efficient for some workloads (e.g., I/O intensive workloads that spend most of their time in the kernel), but CPU intensive workloads will suffer due to the costs of dynamic binary translation and instruction interception.

Once a lock operation is intercepted, DCR logs the value of a per-thread Lamport clock for the acquire operation preceding the memory operation and increments the Lamport clock for the subsequent release operation.

Plane I/O

A key observation behind DCR's recording of control and data plane I/O is that, in its target domain of datacenter applications, the control and data planes can often be manually identified with ease, and if not, automatic methods can be successfully applied. This observation motivates DCR's approach of semi-automatically classifying control-plane I/O. In particular, DCR interposes on communication channels (Section 4.2.1) and then records the ordering and values from only those channels that are semi-automatically classified as control-plane or data-plane channels (Section 4.2.1).

Interposing on Channels. DCR interposes on commonly-used inter-CPU communication channels, regardless of whether these channels connect CPUs on the same node or on different nodes.

Socket, pipe, tty, and file channels are the easiest to interpose efficiently as they operate through well-defined interfaces (system calls). Interpositioning is then a matter of intercepting these system calls, keying the channel on the file-descriptor used in the system call (e.g., as specified in `sys_read()` and `sys_write()`), and observing channel behavior via system call return values.

Shared memory channels are the hardest to interpose efficiently. The key challenge is in detecting sharing; that is, when a value written by one CPU is later read by another CPU. A naive approach would be to maintain per memory-location meta-data about CPU-access behavior. But this is expensive, as it would require intercepting every load and store. One could improve performance by considering accesses to only shared pages. But this too incurs high overhead in multi-threaded applications (i.e., most datacenter applications) where the address-space is shared.

To efficiently detect inter-CPU sharing, DCR employs the page-based Concurrent-Read Exclusive-Write (CREW) memory sharing protocol, first suggested in the context of deterministic replay by Instant Replay [34] and later implemented and refined by SMP-ReVirt [17]. Page-based CREW leverages page-protection hardware found in modern MMUs to detect concurrent and conflicting accesses to shared pages. When a shared page comes into conflict, CREW then forces the conflicting CPUs to access the page one at a time, effectively simulating a synchronized communication channel through the shared page. Details of our CREW implementation are given in Section 6.2.1.

Classifying Channels. Two observations underly DCR's semi-automated classification method. The first is that, in the context of datacenter applications, control plane channels can often be manually distinguished with ease. For example, Hypertable's master and lock server are entirely control plane nodes by design, and thus all their channels are control plane channels. The second observation is that control-plane channels, though bursty, operate at low data-rates (see Section 3). For example, Hadoop job nodes see little communication since they are mostly responsible for job assignment—a relatively infrequent operation.

DCR leverages the first observation by allowing the user to specify or annotate control plane channels. The annotations may be at channel granularity (e.g., all communication to configuration file x), or at process granularity (e.g., the master is a control plane process).

Of course, it may not be practical for the developer to annotate all control plane channels. Thus, to aid completeness, DCR attempts to automatically classify channels. More specifically, DCR uses a channel's data-rate profile, including bursts, to automatically determine whether it is a control plane channel or not. DCR employs a simple token-bucket classifier to detect control plane channels: if a channel does not overflow the token bucket, then DCR deems it to be a control channel; otherwise DCR assumes it is a data channel.

Socket, pipe, tty, and file channels. The token-bucket classifier on these channels are parameterized with a token fill rate of 100KBps and a max size of 1000KBps.

Shared-memory channels. The data-rates here are measured in terms of CREW-fault rate. The higher the fault rate, the greater the amount of sharing through that page. We experimentally derived token-bucket parameters for CREW control-plane communications: a bucket rate of 150 faults per second, and a burst of 1000 faults per second was sufficient to characterize control plane sharing (see evaluation in Chapter 7).

A key limitation of our automated classifier is that it provides only best-effort classification: the heuristic of using CREW page-fault rate to detect control-plane shared-memory communication can lead to false negatives (and unproblematically, false positives), in which case, control plane determinism cannot be guaranteed. In particular, the behavior of legitimate but high data-rate control-plane activity (e.g., spin-locks) will not be captured, hence precluding control-plane determinism of the communicating code. In our experiments,

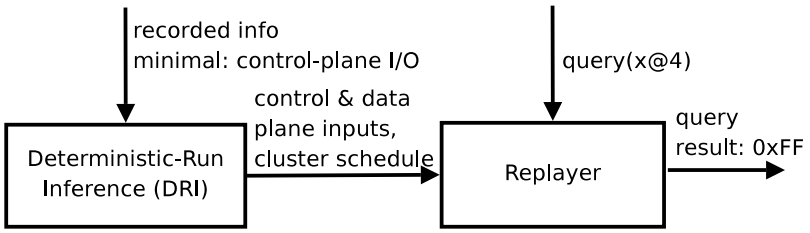

Figure 4.2: An overview of the Distributed Replay Engine (DRE). It uses Deterministic-Run Inference (Chapter 5) to compute unrecorded non-determinism that is then substituted by the Replayer into a new run of the program.

however, such false negatives were rare due to the fact that user-level applications (especially those that use `pthreads`) rarely employ busy-waiting. In particular, on a lock miss, `pthread_mutex_lock()` will await notification of lock availability in the kernel rather than spin incessantly.

4.2.2 Distributed-Replay Engine

The central challenge faced by `DCR`'s Distributed Replay Engine (DRE) is that of providing a control-plane deterministic view of program state in response to analysis queries. This is challenging because, although `DCR` knows the original control-plane inputs, it does not know the original data-plane inputs. Without the data-plane inputs, `DCR` can't employ the traditional replay technique of re-running the program with the original inputs. Even re-running the program with just the original control-plane inputs is unlikely to yield a control-plane deterministic run, because the behavior of the control-plane depends on the behavior of the data-plane.

To address this challenge, the DRE employs Deterministic Run Inference (DRI)–a novel technique for computing the data-plane inputs needed to provide a control-plane deterministic view of program state in response to analysis queries. In particular, DRI leverages the original run's control-plane I/O plus additional information (previously recorded by `DCR`) and program analysis to infer unrecorded non-determinism (e.g., data-plane inputs, thread schedule) that, when substituted along with the original (recorded) control plane inputs into subsequent runs, will induce the original control plane output.

Figure 4.2 gives a closeup view of the DRE, and shows how it uses DRI to answer analysis queries. DRI accepts minimally control plane I/O and then produces the control and data plane inputs as well as the *cluster schedule* (an interleaving of CPUs in the cluster as captured by a Lamport clock) needed to provide control-plane determinism. The resulting information is then fed in to a subsequent replay execution. The requested query state (e.g., the value of a variable) is then extracted from this replay execution once it reaches the query target instruction.

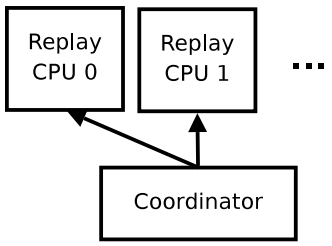

Figure 4.3: The DRE's replayer: a coordinator process schedules replaying CPUs serially per the DRI-produced cluster schedule.

Inference

A detailed treatment of DRI is given in Chapter 5.

Replay

Figure 4.3 gives an overview of the replay stage. It consists of running several replaying CPUs and a coordinator process. Replay proceeds serially: only one CPU executes at a time. The CPU to be executed next is decided by a coordinator process that replays node per the DRI-produced cluster schedule. The replay is done simply by replaying CPU events in increasing Lamport clock order.

Dealing with a Mixed World In an ideal world, all nodes on the network would be using `DCR`. In reality, only some of the nodes (the datacenter application nodes) are traced. External nodes such as the distributed filesystem housing the persistent store and the network (i.e., routers) used by the application are not recorded and hence may behave differently at replay time.

Persistent-Store Nondeterminism. `DCR` does not assume that the persistent store housing data-plane inputs will be recorded and replayed: though this is not prohibited, the persistence hypothesis (that external data-plane input is in persistent storage) does not hold for the persistent-store itself. Indeed, inputs to the persistent-store are rarely accessible (e.g., clicks made by end-users). This poses two challenges for the replayer.

First, replaying applications will need to obtain data plane inputs from the store during replay, but these original inputs may no longer be present on the same nodes at replay time. HDFS, for instance, may redistribute blocks or even alter block IDs. Hence simply reissuing HDFS requests with original block IDs is inadequate. The second challenge is that our target application may use the persistent store to hold temporary files: Hadoop, for instance, stores the results of map jobs to temporary files within an HDFS cluster. Since data read from

these files are part of the data-plane, DCR will not have recorded them. Moreover, the DRE cannot synthesize them because HDFS is not recorded/replayed.

DCR addresses both challenges using a layer of indirection. In particular, DCR requires that the target distributed application communicates with the distributed file-system via a VFS-style (i.e., filesystem mounted) interface (e.g., HDFS's Fuse support or the NFS VFS interface) rather than directly via sockets. The VFS layer addresses the first challenge by providing a well-defined and predictable read/write interface to DCR, keyed only on the target filename, hence shielding it from any internal protocol state that may change over time (e.g., block assignments and IDs). The VFS layer addresses the second challenge by enabling DCR to understand that files are being created and deleted on the DFS. This in turns allows DCR to recreate temporary/intermediate files on the (distributed) file system during replay, in effect re-enacting the original execution.

Network Nondeterminism. DCR does not record and reproduce network (i.e., router) behavior. This introduces two key challenges for DRE.

First, nodes may be replayed on hosts different than those used in the original, making it hard for replaying nodes to determine where to send messages to. For example, DCR's partial replay feature enables a 1000 node cluster to replay on just 100 node if the user so desires, and some of these replay nodes will not receive their original IP addresses. The second challenge is that the network may non-deterministically drop messages (e.g., for UDP datagrams). This means that simply resending a message during replay is not enough to synthesize packet contents at the receiving node: DCR must ensure that the target node actually receives the message.

As with persistent-store non-determinism, DCR shields replaying nodes from network non-determinism using a layer of indirection. That is, rather than send messages through the bare network at replay time, DCR sends messages through REPLAYNET – a virtual replay-mode network that abstracts away the details of IP addressing and unreliable delivery. At the high level, REPLAYNET can be thought of as a database that maps from unique message IDs to message contents. To send a message over REPLAYNET, then, a sending node simply inserts the message contents into the database keyed on the message's unique ID. To receive the message contents, a node queries REPLAYNET with the ID of the message it wishes to retrieve. REPLAYNET guarantees reliable delivery and doesn't require senders and receivers to be aware of replay-host IP addresses.

REPLAYNET poses two key challenges:

Obtaining Unique Message IDs. To send and receive messages on REPLAYNET, senders and receivers must be able to identify messages with unique IDs. These message IDs are simple UUIDs that are assigned at record time. Conceptually, the message ID for each message is logged by both the sender and receiver. The receiver is able to record the message ID since the sender piggy-backs it (using an in-band technique we developed in the liblog system [25])

on the outgoing message at record time. Further details of piggy-backing are given in Section 6.2.3.

Scaling to Gigabytes of In-transit Data. In a realistic datacenter setting, the network may contain gigabytes of in-transit data. Hence, a centralized architecture in which one node maintains the `REPLAYNET` database clearly will not scale. To address this challenge, `REPLAYNET` employs a distributed master/slave architecture in which a single master node maintains a message index and the slaves maintain the messages. To retrieve message contents, a node first consult the master for the location (i.e., IP address) of the slave holding the message contents for a given message ID. Once the master replies, the node can obtain the message contents directly from the slave.

4.2.3 Analysis Framework

A replay system in of itself isn't particularly useful for debugging. That's why `DCR` goes beyond replay to provide a powerful platform for building powerful replay-mode, automated debugging tools. In particular, `DCR` was designed to be extended via plugins, hence enabling developers to write novel, sophisticated, and heavy-weight distributed analyses that would be too expensive to run in production. We've created several plugins using this architecture, including distributed data flow, global invariant checking, communication graph analysis, and distributed-system visualization. Figure 4.4 shows the visualization plugin in action on a replay of the Mesos cluster operating system.

In the remainder of this section we describe `DCR`'s plugin programming model, and then demonstrate its power with a simple automated-debugging plugin: distributed data flow analysis.

Programming Model

`DCR` plugins are written in the Python programming language, which we selected for its ease of use and suitability for rapid prototyping. A key goal of `DCR`'s plugin architecture is to ease the development of sophisticated plugins. Toward this goal, `DCR` plugin model provides the following properties, many of which are borrowed from the Friday distributed replay system [24]:

An illusion of global state. `DCR` enables plugins to refer to remote application state as though it was all housed on the same machine. For example, the following code snippet grabs and prints a chunk of memory bytes from node id 2:

```
my_bytes = node[2].mem[0x1000:4096]
print my_bytes
```

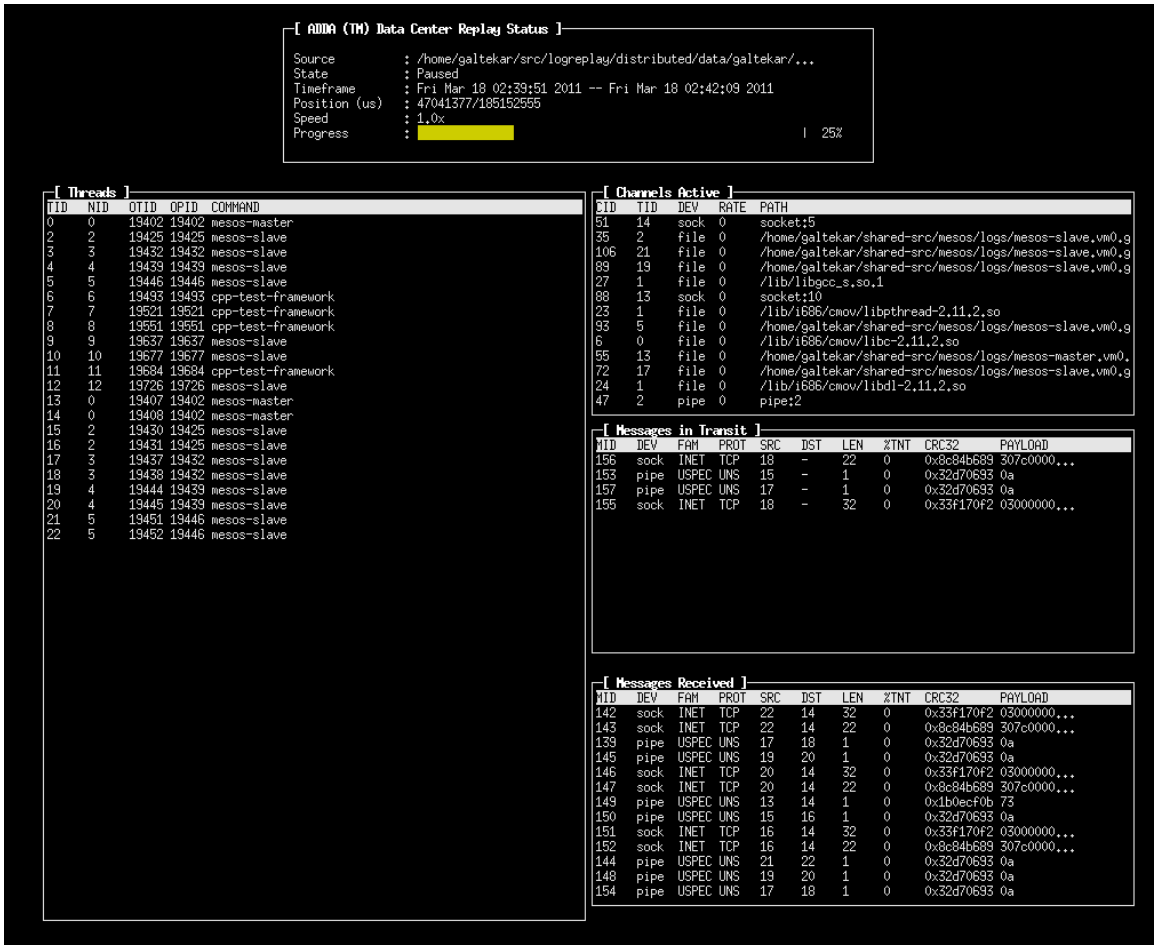

Figure 4.4: A DCR visualization plugin running on a replay of the Mesos cluster operating system. This ncurses-based plugin provides a dynamic, birds-eye view of the distributed system in replay, including active threads/nodes, open communication channels (sockets, pipes, files), and in-transit/received messages.

The key point is that DCR abstracts away the details of messaging.

An illusion of serial replay. DCR guarantees that plugin execution is serializable and deterministic, hence freeing the plugin-developer from having to reason about concurrency and potentially non-deterministic plugin results. For example, the following plugin (initialization code omitted) is guaranteed to deterministically assign a total ordering to all received messages in a replay execution:

```
i = 0
def on_recv(msg):
   print i, msg
   i += 1
```

Access to fine-grained analysis primitives. Plugin developers shouldn't have to reinvent the wheel. Thus, DCR comes preloaded with commonly-used, powerful, fine-grained analysis primitives. An example of such a primitive is DCR's data-flow analysis primitive. The primitive exports two key functions (`is_tainted(node, addr)` and `set_taint(node, addr)`) that plugins can invoke to determine if some origin data has influenced and to taint the byte at `addr`, respectively. Other primitives include lock-tracing, race-detection, and formula generation (whereby one may obtain the precise logical relationships between between program state). These primitives require introspecting replay execution at instruction level, which DCR does using binary translation (discussed in more detail in Section 6.3.2).

4.3 Related Work

Much work has been done on deterministic replay, but none of the prior work focused on replaying production datacenter applications or addressed the technical challenges raised by that set of requirements (see Section 1.1). In particular, DCR's support for efficiently recording a large number of commodity machines running a broad-range of terabyte-sale, data-intensive software is new, and it is the first to support efficient recording of multiprocessor-enabled applications, including those with data races.

More specifically, Figure 4.5 compares DCR with other replay-debugging systems along key dimensions. The following paragraphs explain why existing systems do not meet our requirements.

Always-On Operation. Classical replay systems such as Instant Replay, liblog, VMWare, and SMP-ReVirt are capable of, or may be easily adapted for, large-scale distributed operation. Nevertheless, they are unsuitable for the datacenter because they record all inbound disk and network traffic. The ensuing logging rates, amounting to petabytes/week across all datacenter nodes, not only incur throughput losses, but also call for additional storage infrastructure (e.g., another petabyte-scale DFS).

	Always-on operation	Whole-cluster replay	Wide applicability	Determinism
Hardware support [15, 35, 39], CoreDet [9]	*No*	*No*	*No*	Value
liblog [25], VMWare [6], PRES [43], ReSpec [36]	*No*	*No*	Yes	I/O
SherLog [50]	Yes	*No*	*No*	Output
ODR [7], ESD [52]	Yes	*No*	Yes	Output
Instant Replay [34], DejaVu [32]	Yes	*No*	Yes	Value
R2 [27]	Yes	Yes	*No*	Value
DCR	**Yes**	**Yes**	**Yes**	**Control**

Figure 4.5: A comparison with other replay-debugging systems. Only DCR meets all the requirements for *datacenter applications*.

Several relaxed-deterministic replay systems (e.g., Stone [46], PRES [43], and ReSpec [36]) and hardware and/or compiler assisted systems (e.g., Capo [39], Lee et al. [35], DMP [15], CoreDet [9], etc.) support efficient recording of multi-core, shared-memory intensive programs. But like classical systems, these schemes still incur high record-rates on network and disk intensive distributed systems (i.e., datacenter systems).

Whole-Cluster Replay. Several replay systems can provide whole-cluster replay for small clusters, but not for large-scale, failure-prone datacenters. Specifically, systems such as liblog [25], Friday [24], VMWare [6], Capo [39], PRES [43], and ReSpec [36] allow an arbitrary subset of nodes to be replayed, but only if recorded state on that subset is accessible. Order-based systems such as DejaVu and MPIWiz may not be able to replay even a small subset of nodes in the event of node failure, because nodes rely on message senders to regenerate inbound messages during replay.

Recent output-deterministic replay systems such as ESD [52] and SherLog [50] can efficiently record and replay single-node multi-threaded applications using an inference technique similar to that of `DCR`. However, unlike `DCR`, these systems (and their respective inference techniques) were not designed for distributed operation, and thus do not support whole-cluster operation. Thus the techniques presented in this disseration are largely Note that we do not claim to be superior to these systems, and we believe that their inference techniques are in many ways complementary to those used in `DCR` (e.g., the use of static analysis to reduce inference effort).

Wide Applicability. Several replay schemes employ hardware support for efficient multiprocessor recording. These schemes don't address the problem of efficient datacenter recording, however. In particular, they still incur significant slowdowns (2x) when recording I/O-intensive computations. What's more, they currently exist only in simulation, so they don't meet our commodity hardware requirement.

Single-node, software-based systems such as CoreDet [9], ESD [52], and SherLog [50] employ source-code analyses to speed the inference process. In contrast, `DCR` is language and runtime indifferent, since it operates at the instruction level (using a combination of binary translation and system call interception). This indifference enables `DCR` to work even on datacenter applications with blackbox components (e.g., a third-part library).

The R2 system [27] provides an API and annotation mechanism by which developers may select the application code that is recorded and replayed. Conceivably, the mechanism may be used to record just control-plane code, thus incurring low recording overheads. Alas, such annotations require considerable developer effort to manually identify data-races (hard even for automated techniques) and the control-plane, and to retrofit existing code bases.

Chapter 5

Deterministic-Run Inference

The technical centerpiece of this work is Deterministic-Run Inference (DRI)—an offline program analysis that uses recorded information and a datacenter application's code to *compute* the unrecorded non-determinism of a control-plane deterministic run. It constitutes the heart of `DCR` and we describe its key concepts in Section 5.1. Like many inference-based program analyses, the central challenge with DRI is in making it work with realistic programs. We developed several optimizations that, in many instances, reduce the inference complexity by exponential quantities. Section 5.2 describes these optimizations in further detail.

5.1 Concept

The key concept behind DRI is the notion of a *determinism condition*. With an abuse of notation, we may easily represent a determinism condition as,

$$f(c_i, d_i, t) = (c_o, d_o)$$

where formula f is a quantifier-free first-order logic translation of the entire distributed program and $=$ denotes a restriction on the program outputs. In short, the determinism condition expresses a distributed program's control and data plane output (c_o and d_o, respectively) as a function of its control plane inputs c_i, data plane inputs d_i, and thread schedule t. DRI is given concrete values for c_i and c_o as `DCR` always record them (see Chapter 4 for more details on what `DCR` records and how). However, d_i, d_o, and t may not have been recorded and are therefore the unknowns (i.e., variables) in the formula. Any satisfiable solution to this formula, then, gives the non-deterministic information needed to produce a control-plane deterministic run.

Leveraging this observation, DRI works in two stages. In the first stage, *determinism condition generation*, DRI translates the distributed program into a *determinism condition*—a first-order logic formula whose variables represent unrecorded non-determinism (e.g., data-plane inputs and thread schedule). Section 5.1.1 describes the generation process in detail. In

the second phase, *determinism condition solving*, DRI dispatches the determinism condition generated in the previous phase to an automated constraint solver. The solver computes a satisfiable assignment of values to variables in the formula, thereby instantiating a control-plane deterministic run. Section 5.1.2 describes the solving process in further detail.

5.1.1 Determinism Condition Generation

A determinism condition generator takes a program, an execution trace, and a count of threads in the trace to produce a determinism condition. To perform this translation, the generator employs cluster symbolic execution—the distributed variant of a well-known single-node program verification technique [31]. This section details this translation procedure on a simple cluster programming model and language. We first describe this model and language, define what a determinism condition generator is in the context of this model, and conclude with the mechanics and evaluation rules for cluster symbolic execution.

Cluster Computation Model and Language

To simplify presentation, and without loss of generality, we present determinism condition generation (i.e., formula generation) in the context of a simple concurrent computation model and language.

Setup. The cluster is assumed to consist of at least one node, and at least one node is assumed to have been traced by `DCR`—we call these *traced nodes*. We call the remaining nodes *untraced nodes*. All traced nodes are assumed to execute the same program.

The cluster consists of a fixed number of execution threads, distributed among all CPUs (the precise assignment of threads to CPUs may be arbitrary—each CPU may be assigned just one thread, for example). Multi-thread execution is guaranteed to be sequentially consistent.

Communication between threads takes place along *channels*—an abstraction of the shared memory and messaging passing channels typically for inter-thread communication. If a channel connects two traced nodes, we say that it is an *internal channel*. But if it connects a traced and untraced node, we say that it is a *boundary channel*. In the subsequent presentation, we will use the IsBoundary() predicate to distinguish internal and boundary channels.

Channels operate in a FIFO fashion and provide one word for in-transit storage; data enters and leaves a channel one word at a time and atomically. Reading from an empty channel returns a fixed value (e.g., 0). Writing to a filled channel results in the existing value being overwritten. Details of connection setup and shutdown are abstracted away: a fixed but sufficient number of channels is assumed to exist between any two threads.

Language (CIMP). All traced nodes execute a program written in Cluster Imperative Programming Language, or CIMP for short. CIMP has the following assembly-like syntax:

$$\text{x := e} \mid \text{if e goto L} \mid \text{L:} \mid \text{x := read}_c() \mid \text{write}_c(\text{x}) \mid \text{exit} ,$$

where x corresponds to a variable in program state, e denotes an expression that references program state, := denotes an assignment and "L:" denotes an instruction label L.

To elaborate, the x := e command results in an evaluation of expression e followed by an assigned to variable x (i.e., write into the memory location that hosts variable x). The if e goto L statement represents a conditional branch to label L. The $\text{read}_c()$ statement denotes a read from communication channel c while $\text{write}_c(x)$ denotes a write of variable x into communication channel c. The exit statement terminates execution.

CIMP programs may have multiple yet a fixed and predefined number of threads, each of which has a unique identifier (i.e., a thread identifier). Threads also have their own set of variables. A CIMP thread may not directly modify the variables of another CIMP thread (even if it is running on the same node); such modification must be done by message-passing on channels. While CIMP does not support shared memory communication for simplicity of presentation, it may be built on top of the supported message passing primitives.

Execution Model. Executing a CIMP program entails executing instructions from each CIMP thread one by one (i.e., in a serial manner) on a global *cluster state*

$$\sigma : \text{Var} \rightarrow \text{Value},$$

where Var denotes the set of all variables across all nodes in the distributed execution and Value denotes a single word-sized value.

As a shorthand, we will use $\sigma(x_t)$ to denote the value of variable x on thread t, while $\sigma[x_t := e]$ denotes a new state in which x's value is now e but identical to the old state everywhere else. We say that $\sigma_i \subseteq \sigma_j$ iff $\sigma_i(x) = v \Rightarrow \sigma_j(x) = v$ for all x.

Some variables in the cluster state are predefined and have special meaning. In particular, $\sigma(pc_t)$ denotes the program counter value of thread t, and is updated automatically after each instruction is executed. Though it can never be explicitly altered by a CIMP program, we need it to identify the instruction about to be executed when describing evaluation techniques. Also $\sigma(i)$ denotes the value of a global instruction count (i.e., shared among all threads) that is automatically incremented after every instruction execution, while $\sigma(b)$ denotes the values of a global branch count incremented after ever branch instruction. The value $\sigma(ch_c)$ denotes the in-transit datum in communication channel c.

Program *execution* E, then, is a sequence of cluster states $\sigma_0, \sigma_1, \ldots, \sigma_n$, where each new state results from the execution of an instruction from a single arbitrarily chosen thread. Thus, all valid CIMP executions are sequentially consistent. Execution of a thread terminates (i.e., the thread dies) when it issues an `exit` instruction. Cluster execution stops when all threads have executed an `exit` instruction. As shorthand, we will use E_i to refer to σ_i.

Definition of Determinism Condition Generator

We begin with a general definition of what it means for an execution to be deterministic. Then we define a determinism condition generator using this definition.

Trace of Execution E. A sequence of cluster states $r_0, r_1, \ldots, r_n$ such that for all r_i it holds that $r_i \subseteq E_i$.

$\mathcal{D}$-Deterministic Execution. An execution $\sigma_0, \sigma_1, \ldots, \sigma_n$ that has the same length as trace $\mathcal{D}$, and for all σ_i, it holds that $\mathcal{D}_i \subseteq \sigma_i$. The precise brand of determinism then depends entirely on how $\mathcal{D}$ is defined. For example, the traditional definition of deterministic execution may be describe by letting $\mathcal{D}$ denote a trace of all I/O, hence resulting in a I/O-deterministic execution.

$\mathcal{D}$-Determinism Condition Generator. A $\mathcal{D}$-determinism condition generator $\mathcal{G}_\mathcal{D}$ is a function that maps some original execution trace $\mathcal{D}$ of CIMP program P, and a set of threads to a quantifier-free first-order logic assertion that represents the set of all $\mathcal{D}$-deterministic runs of the program. More precisely,

$$\mathcal{G}_\mathcal{D} : \mathcal{D} \times P \times T \rightarrow \text{Assertion}.$$

While many kinds of determinism condition generators are possible (depending on the definition of $\mathcal{D}$), `DCR` aims for one particular kind—a control-plane determinism condition generator (see Chapter 3 for an introduction to control-plane determinism).

Control-Plane Determinism Condition Generator. A control-plane determinism condition generator $\mathcal{G}_\alpha$ is a α-determinism condition generator, where α is *minimally* an execution trace of the original run's control plane I/O. Specifically, if it holds that the instruction to be executed in state i of the original execution is a read or write of a control plane I/O channel into or from variable x (respectively), then it holds that x is defined in α_{i+1} and maps to the read or written value. We emphasize minimality because the control-plane determinism property is not lost if the trace contains more information than just the control plane I/O.

Generation Method

DRI generates a determinism condition using a variant of classic Floyd-style verification condition generation (VCGEN) [22]. The key idea is to interpret program code and translate each statement into a logical constraint. Interpretation is done via symbolic execution [31] which proceeds much like normal (i.e., concrete) execution. However, the key difference is that instructions operate on symbolic state—state in which data is represented as expressions on program input rather than a single (i.e., concrete) value. As symbolic execution proceeds,

the verification condition generator outputs a logical constraint on program state for select program statements, thus limiting the set of concrete states the symbolic state represents to those consistent with a control-plane deterministic run.

While verification condition generation has been well-studied in the context of single-threaded, single-node programs , generation for multi-threaded, multi-node (i.e., distributed) programs remains difficult. In particular, concurrency (distributed across a datacenter or on a single node) introduces two challenges. First, concurrent systems may be composed of multiple threads (potentially thousands, spread across many nodes), and the condition must capture all of their behaviors. And second, the behavior of any given thread in the system may depend on the behavior of other threads. Thus the resulting condition needs to capture the collective behavior (i.e., inter-thread interactions) of the system so that inferences DRI makes from the formula are causally consistent across threads.

Cluster Symbolic Execution (CSE). DRI's determinism condition generator addresses both requirements using a technique we call *cluster symbolic execution* (CSE). CSE differs from traditional symbolic execution in two key ways. First, it performs symbolic execution for each thread and along multiple thread schedules, hence generating a formula that represents executions of all threads in the cluster. Second, it employs the notion of a cluster symbolic state to restrict the set of executions to those that are causally consistent (i.e., those in which received messages were previously sent), hence capturing inter-thread interactions.

Cluster symbolic execution proceeds much like traditional symbolic execution, except that threads act on a global *cluster symbolic state*

$$\Sigma : \text{Var} \rightarrow \text{SymbolicExpr},$$

where Var denotes the set of all variables across all nodes in the distributed execution (i.e., all cluster state) and SymbolicExpr denotes the set of all symbolic expressions. For simplicity of discussion (but without loss of generality), we limit the language of symbolic expressions to the form

$$\text{Expr} := \text{Expr} + \text{Expr} \mid s_i \mid \text{Const},$$

where Expr denotes a symbolic expression, s_i denotes the i-th symbolic variable (which represents an unknown data value), and Const denotes a concrete value.

As a shorthand, we will use $\Sigma_i(x_t)$ to denote the symbolic expression of variable x on thread t in symbolic state index i, while $\Sigma_i[x_t := e]$ denotes a new symbolic state in which variable x_t's value is now e but identical to symbolic state i everywhere else (i.e., the result of an assignment). We use the expression $\Sigma_i[x_t \mapsto_+ 1]$ to denote a new state in which x_t is incremented by 1, or more precisely $\Sigma_i[x_t \mapsto \Sigma(x_t) + 1]$.

Special variables. The variables pc_t, ch_c, b, i in the symbolic state are read-only and have special meaning. In particular, $\Sigma(pc_t)$ denotes the program counter value of thread t. It is never explicitly modified by the program, and is always concrete. Similarly, $\Sigma(ch_c)$ denotes

the in-transit symbolic value of communication channel c. It can be accessed only via channel read and write primitives, and may map to a symbolic value. $\Sigma(b)$ denotes a read-only count of the number of branches executed by all threads in the program, and $\Sigma(i)$ denotes an instruction count incremented after every executed instruction. Both b and i are always concrete.

Bookkeeping map. A bookkeeping map is a function B : SpecialVar $\rightarrow$ Const, where SpecialVar denotes a set of program-inaccessible variables used for bookkeeping purposes during symbolic execution. More precisely, SpecialVar $= \{v\}$, where v denotes a symbolic variable counter incremented when a fresh symbolic variable is needed.

Generating a Determinism Condition Using CSE. *$\mathcal{D}$-guided cluster symbolic evaluator.* We define this as a function,

$$E : \mathcal{D} \times P \times \mathcal{P}(T) \times T \times \Sigma \times B \rightarrow \text{Assertion},$$

where $\mathcal{D}$ denotes an original execution trace of CIMP program P, T denotes the set of CIMP thread identifiers, Σ denotes a cluster symbolic state, B denotes a bookkeeping map, and Assertion denotes the set of quantifier-free first-order logic expressions. As a shorthand, we use $E_{\mathcal{D},P,T}$ to denote E for some fixed $\mathcal{D}, P, \mathcal{P}(T)$.

Figure 5.1 gives the symbolic evaluation rules for a simple control-plane deterministic generator. This $\mathrm{E}_{\alpha,P,T}$ evaluator compares and contrasts with existing single-threaded symbolic evaluators used in the program verification and testing domains in the following ways.

Inputs. The $\mathrm{E}_{\alpha,P,T}$ evaluator treats all words read from boundary data-plane channels as symbolic inputs: it assigns a fresh symbolic value s_i to each incoming word. In contrast, reads from control-plane channels (boundary or internal) are concretized using the original values as found in the trace α, where the concretization is performed by simply assigning the incoming concrete value to the target variable.

The selective concretization performed by $\mathrm{E}_{\alpha,P,T}$ contrasts with traditional non-cluster symbolic evaluators used in the software testing and verification domain where all user-inputs (e.g., keyboard input, incoming network message), control or data, are assigned fresh symbolic variables, and where no effort is made to distinguish between inputs originating from traced nodes and untraced nodes.

Outputs. On control plane writes, the $\mathrm{E}_{\alpha,P,T}$ evaluator emits constraints that bind the output to the original traced control-plane outputs. Existing symbolic evaluators typically don't restrict the outputs. Instead, they typically generate constraints on the path taken by the program. Note that $\mathrm{E}_{\alpha,P,T}$ does not constrain the output of data plane channels. It cannot because it does not know what the original data plane outputs were–they were never recorded. Existing symbolic evaluators typically don't distinguish between control and data

$$\mathrm{E}_{\alpha,P,T}(t,\Sigma,B) =$$

$$\begin{cases}
\mathrm{E}_{\alpha,P,T}(t,\Sigma[x \mapsto \Sigma(e), pc_t \mapsto_+ 1, i \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := e}, \\
\bigvee_{t' \in T} \mathrm{E}_{\alpha,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto \alpha_{\Sigma(i)+1}(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsControl}(c) \\
\bigvee_{t' \in T} \mathrm{E}_{\alpha,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto ch_c], B]) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsData}(c) \wedge \neg\mathrm{IsBoundary}(c) \\
\bigvee_{t' \in T} \mathrm{E}_{\alpha,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto s_{B(v)}], B[v \mapsto_+ 1]) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsData}(c) \wedge \mathrm{IsBoundary}(c) \\
\Sigma(x) = \alpha_{\Sigma(i)}(x) \wedge \bigvee_{t' \in T} \mathrm{E}_{\alpha,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsControl}(c), \\
\bigvee_{t' \in T} \mathrm{E}_{\alpha,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsData}(c), \\
e \Rightarrow \mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto L, i \mapsto_+ 1, b \mapsto_+ 1], B) \wedge \neg e \Rightarrow \mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, b \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) < \alpha_n(b), \\
\mathrm{False} & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) = \alpha_n(b), \\
\mathrm{True} & \text{if } P_{\Sigma(pc_t)} = \texttt{exit}.
\end{cases}$$

Figure 5.1: Symbolic evaluation rules for generating a control-plane determinism condition for control-plane I/O trace α, thread identifiers T, and CIMP program P. Evaluation of these rules will result in a logical formula representing the set of all control-plane deterministic runs starting with symbolic state Σ and bookkeeping state B.

plane outputs.

Branches. The $E_{\alpha,P,T}$ evaluator considers execution along both paths of a branch, and thus, the resulting formula represents execution along all multiple execution paths. Specifically, if the branch is taken then e holds and symbolic evaluation continues at the branch target L. However, if the branch is not taken then e does not hold and evaluation continues at the fall-through instruction.

This generation strategy is similar to traditional verification condition generators (e.g., that in PCC). However, it is unlike that used in testing, where per-path formulas are iteratively generated and solved, hence often avoiding the need to search an exponential space of paths before converging on to an answer. We note that similar optimizations are also possible with $E_{\alpha,P,T}$, but we chose an all-paths formula generation phase for simplicity of presentation.

Note however, that $E_{\alpha,P,T}$ doesn't specify the path exploration strategy. Here multiple backtracking techniques are possible (e.g., DFS vs. BFS) or the fork() model (as done by EXE), each with their respective tradeoffs.

Loops and Termination. At some point, symbolic evaluation must terminate. The key challenge here is loop, which may not terminate. We handle this by upper-bounding the number of branches explored to $\alpha_n(b)$, which denotes the number of branches originally executed. Bounding the path exploration is sound, since there exists some control-plane deterministic run that executes at most the original number of branches—the original run is a witness.

Symbolic evaluation of loops has been extensively studied and many techniques have been used to ensure termination (and polynomial runtime) of formula generation. For example, PCC leverages loop invariant annotations to preclude exponential blowup in path-space exploration. $E_{\alpha,P,T}$ does not leverage loop invariants, though there is nothing precluding their use beyond the annotation burden. An alternative strategy is to limit the number of loop iteration to a fixed quantity, as done by ESC-Java [21]. But the resulting formula may not represent all (or any) control-plane deterministic executions, and hence is unsound.

5.1.2 Determinism Condition Solving

To solve a determinism condition, DRI sends the formula to a constraint solver. We use the STP solver [23] because it supports bitvector arithmetic—an essential property for reasoning about code at the instruction level. Once solved, STP provides a concrete assignment of values to variables in the input formula.

5.2 Taming Intractability

The chief difficulty with the base DRI presented in Section 5.1 is that it scales poorly to large applications. The main reason is that inference entails what is essentially a search over all possible bounded-length executions (where the bound is the original run's branch count). More specifically, inference involves searching over two exponential-size spaces: the space of all bounded-length inputs and the space of all bounded-length thread-schedules. Here we present techniques for pruning each of these search spaces.

5.2.1 Input-Space Reduction

We present two techniques for reducing the search space of inputs. The key idea behind both is to use knowledge of original run to limit the size of the determinism condition. Section 5.2.1 presents input guidance, while Section 5.2.1 presents path guidance.

Input Guidance

The key idea behind input guided determinism condition generation is to consider only those control-plane deterministic runs consistent with the original inputs (data as well as control inputs). Input guidance is sound in that it is guaranteed to produce a formula representing at least one control-plane deterministic run. This holds because we know there exists a control-plane deterministic run that obtains exactly those inputs—the original run.

Figure 5.2 gives the evaluation rules for input guided formula generation. The rules differ from the naive rules (as given in Figure 5.1) in two ways. First, the input guidance rules assume access to an execution trace β that includes data-plane inputs as well as control plane I/O. The second key difference is that the β trace is used to concretize all data, hence doing away with symbolic variables in the resulting formula altogether.

Input guidance confers two key benefits. First, for any given thread schedule, $E_{\beta,P,T}$ need only consider one program path. Since all inputs are known, branch conditions are concrete and their outcomes are may be evaluated during symbolic evaluation rather than at formula solving time. The second advantage is that a determinism condition resulting from input guidance results in formula solving time that is polynomial with formula size. This is due to the fact that no backsolving of the formula is necessary: it doesn't have any unknowns. The key drawback of input guidance is the fact that DRI needs the original input to use it, and recording the original inputs can be costly in the datacenter environment.

Path Guidance

The key idea behind path-guided determinism condition generation is to consider only those control-plane deterministic runs consistent with the original program path (i.e., a trace of branch conditions). Path guidance is sound in that it is guaranteed to produce a formula

$$\mathrm{E}_{\beta,P,T}(t,\Sigma,B) =$$

$$\begin{cases}
\mathrm{E}_{\beta,P,T}(t,\Sigma[x \mapsto \Sigma(e), pc_t \mapsto_+ 1, i \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := e}, \\
\bigvee_{t' \in T} \mathrm{E}_{\beta,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto \beta_{\Sigma(i)+1}(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsControl}(c) \\
\bigvee_{t' \in T} \mathrm{E}_{\beta,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto ch_c], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsData}(c) \wedge \neg\mathrm{IsBoundary}(c) \\
\bigvee_{t' \in T} \mathrm{E}_{\beta,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto \beta_{\Sigma(i)}(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsData}(c) \wedge \mathrm{IsBoundary}(c) \\
\Sigma(x) = \beta_{\Sigma(i)}(x) \wedge \bigvee_{t' \in T} \mathrm{E}_{\beta,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsControl}(c), \\
\bigvee_{t' \in T} \mathrm{E}_{\beta,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsData}(c), \\
\mathrm{E}_{\beta,P,T}(t,\Sigma[pc_t \mapsto L, i \mapsto_+ 1, b \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) < \beta_n(b) \wedge \Sigma(e), \\
\mathrm{E}_{\beta,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, b \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) < \beta_n(b) \wedge \neg\Sigma(e), \\
\mathrm{False} & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge b = \beta_n(b), \\
\mathrm{True} & \text{if } P_{\Sigma(pc_t)} = \texttt{exit}.
\end{cases}$$

Figure 5.2: Symbolic evaluation rules for input-guided formula generation. A control and data plane input (i.e., all input) trace β is used to guide formula generation along one program path. The resulting formula will not have any symbolic variables as the original program input values are assumed to be known.

$$\mathrm{E}_{\gamma,P,T}(t,\Sigma,B) =$$

$$\begin{cases}
\mathrm{E}_{\gamma,P,T}(t,\Sigma[x \mapsto \Sigma(e), pc_t \mapsto_+ 1, i \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := e}, \\
\bigvee_{t' \in T} \mathrm{E}_{\gamma,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto \gamma_{\Sigma(i+1)}(x)], B[r \mapsto B(r)+1]) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsControl}(c) \\
\bigvee_{t' \in T} \mathrm{E}_{\gamma,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto ch_c], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsData}(c) \wedge \neg\mathrm{IsBoundary}(c) \\
\bigvee_{t' \in T} \mathrm{E}_{\gamma,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto s_{B(v)}], B[v \mapsto_+ 1]) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsData}(c) \wedge \mathrm{IsBoundary}(c) \\
\Sigma(x) = \gamma_{\Sigma(i)}(x) \wedge \bigvee_{t' \in T} \mathrm{E}_{\gamma,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsControl}(c), \\
\bigvee_{t' \in T} \mathrm{E}_{\gamma,P,T}(t',\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsData}(c), \\
\Sigma(e) = \gamma_{\Sigma(i)}(e) \wedge \mathrm{E}_{\gamma,P,T}(t,\Sigma[pc_t \mapsto L, i \mapsto_+ 1, b \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) < \gamma_n(b) \wedge \gamma_{\Sigma(i)}(e), \\
\Sigma(e) = \gamma_{\Sigma(i)}(e) \wedge \mathrm{E}_{\gamma,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, b \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) < \gamma_n(b) \wedge \neg\gamma_{\Sigma(i)}(e), \\
\mathrm{False} & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge b = \gamma_n(b), \\
\mathrm{True} & \text{if } P_{\Sigma(pc_t)} = \texttt{exit}.
\end{cases}$$

Figure 5.3: Symbolic evaluation rules for path-guided formula generation. A path trace γ is used to guide the symbolic execution along the original program path.

representing at least one control-plane deterministic run. This holds because we know there exists at least one control-plane deterministic run that takes the given path—the original run.

Figure 5.3 gives the evaluation rules for path-guidance. These rules differ from the naive evaluation rules in two respects. First, they use a trace that includes the original program paths in addition to the control I/O. The second difference is that symbolic evaluation is forced along the path that was originally taken, as reflected in the branch rule.

The key benefit of directed path execution is that it allows formula generation to proceed along just one path rather than an exponential number of them. The key drawback is that it requires knowledge of the original run's path, which may be prohibitive to trace in its entirety (though tracing a partial path may be efficient).

5.2.2 Schedule-Space Reduction

We present techniques for pruning the search space of schedules. We begin with partial-order guidance, a technique that prunes schedules that are unlikely to result in control plane deterministic runs. Taken to its extreme, partial-order guidance can eliminate the need to search multiple thread schedules. We also present consistency relaxation, a technique that at its extreme also eliminates the needs to explore multiple schedules.

Partial-Order Guidance

The key idea behind partial-order guided formula generation is to consider only those program runs consistent with a partial ordering of instructions in the original run. Partial order guidance is sound in that it is guaranteed to produce a formula representing at least one control-plane deterministic run. This holds because we know there exists at least one such run that is consistent with the original partial ordering—the original run. We begin this section with some basic definitions and then give evaluation rules for two variants of partial-order guidance: communication-order guidance and lock-order guidance.

Definitions. A *partial ordering of instructions* Γ is a partially ordered set where we say that $(a_i, b_j) \in \Gamma$, or equivalently $a_i \rightarrow_\Gamma b_j$, if the i-th dynamic instruction executed by thread a was traced as having executed before the j-th dynamic instruction executed by thread b. A *linearization of set* Γ denoted L_Γ is some total ordering consistent with the partial order set Γ. We define this linearization as a function $L_\Gamma : \text{InsnCount} \rightarrow \text{ThreadID}$, such that $L_\Gamma(i) = t$ indicates that the thread with identifier t executed the i-th instruction in this linearization.

Evaluation Rules. Figure 5.4 gives the evaluation rules for a partial-order guided generator $\mathcal{O}_{\alpha,\Gamma,P,T}$, where α denotes a trace of the original control plane I/O, and L_Γ is a

$$\mathcal{O}_{\alpha,\Gamma,P,T}(t,\Sigma,B) =$$

$$\begin{cases}
\mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[x \mapsto \Sigma(e), pc_t \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := e}, \\
\mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[pc_t \mapsto_+ 1, x \mapsto \alpha_{\Sigma(i+1)}(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \text{IsControl}(c) \\
\mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto ch_c], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \text{IsData}(c) \wedge \neg\text{IsBoundary}(c) \\
\mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto s_{B(v)}], B[v \mapsto_+ 1]) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \text{IsData}(c) \wedge \text{IsBoundary}(c) \\
\Sigma(x) = \alpha_{\Sigma(i)}(x) \wedge \mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \text{IsControl}(c), \\
\mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, ch_c \mapsto \Sigma(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \text{IsData}(c), \\
e \Rightarrow \mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[pc_t \mapsto L, i \mapsto_+ 1, b \mapsto_+ 1], B) \wedge \\ \neg e \Rightarrow \mathcal{O}_{\alpha,\Gamma,P,T}(L_\Gamma(i+1),\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, b \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) < \alpha_n(b), \\
\text{False} & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) = \alpha_n(b), \\
\text{True} & \text{if } P_{\Sigma(pc_t)} = \texttt{exit}.
\end{cases}$$

Figure 5.4: Symbolic evaluation rules for partial-order guidance, where α denotes a control-plane I/O trace (as in the native generation rules). Only one instruction schedule is explored per the linearization L_Γ.

linearization of some user-defined partial ordering of instructions in the original run. Unlike the base evaluation rules, $\mathcal{O}_{\alpha,\Gamma,P,T}$ explores just one thread schedule, as determined by L_Γ.

The benefits of partial-order guidance depend on the particular partial ordering that was traced. For example, the original total order of instruction execution (also a partial ordering by definition) reduces the search space to just one schedule (the original total ordering), but it's not practical because it is prohibitive to collect (since it requires per-access instrumentation). The following paragraphs present two definitions of Γ that are practical to trace yet provide a reduction in the schedule space that is close to, if not as good as, having a total order trace.

Communication-Order Guidance. The key observation behind communication-order guidance is that threads influence each others' behaviors only via channel communication (i.e., reads and writes to channels). This observation then implies that one needn't explore all total orderings of instructions. Instead, it suffices to explore just one total ordering that is consistent with the original ordering of channel communication. More precisely, define the original ordering of channel communication as

$$\Gamma = \{(a_i, b_j) \mid \text{SameChannel}(a_i, b_j) \wedge \text{OccurredBefore}(a_i, b_j)\},$$

where $\text{SameChannel}(a_i, b_j)$ holds iff at least one of a_i and b_j is a write and the channel accessed by a_i was the same as that accessed by b_j. $\text{OccurredBefore}(a_i, b_j)$ holds iff a_i occurred before b_i in the original run. Now define a linearization L_Γ as some total ordering consistent with the partial order Γ. Then the evaluator in $\mathcal{O}_{\alpha,\Gamma,P,T}$ (see Figure 5.4) will generate a communication-order guided formula.

Lock-Order Guidance. The key idea behind lock-order guidance is to explore only those schedules consistent with the original ordering of locking operations, where we define a locking operation as a channel read or write that is used for inter-thread synchronization. Using the lock-order rather than the communication order is beneficial for two reasons. First, the lock order is more compact than the communication order, as it is a small subset of all communications in a typical program. Second, the lock order implicitly captures the ordering of most, if not all, channel communication, and therefore provides a schedule-space reduction that is often comparable to communication-order guidance.

Figure 5.5 demonstrates the benefit of lock-order guidance with a few examples. In data-race free programs, knowing the lock order also tells us the ordering of conflicting accesses protected by the lock. Thus exploring a linearization of the lock order is equivalent to exploring a linearization of the communication order. On the other hand, in executions with data races, the lock order doesn't tell us about the ordering of all conflicting accesses (specifically, the racing accesses), but it still narrows down the possibilities. In Figure 5.5(b), for example, we would need to explore two schedules: one in which $read(1, x) \rightarrow write(1, x)$ and another in which $write(1, x) \rightarrow read(1, x)$.

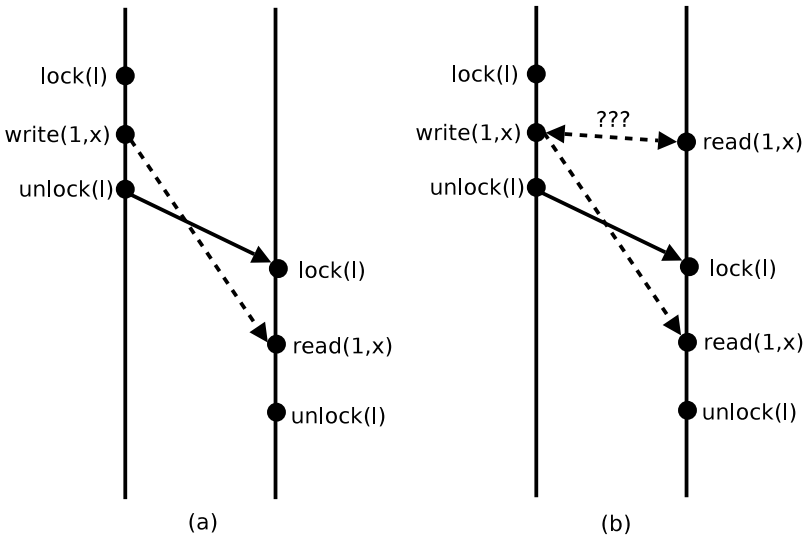

Figure 5.5: (a) The lock order (solid) gives us the communication order (dashed) in data-race free runs. (b) But multiple schedules must still be explored when there are data-races, since lock order does not order such accesses.

To see how lock order guidance fits with the evaluation rules in Figure 5.4, we define the original ordering of lock operations as a partially ordered set

$$\Gamma = \{(a_i, b_j) \mid \text{LockOp}(a_i) \wedge \text{LockOp}(b_j) \wedge \text{OccurredBefore}(a_i, b_j)\},$$

where LockOp predicate holds true if instruction a_i is a inter-communication used for acquiring or releasing a lock. Now define the set of all linearizations of the lock order Γ as $\mathcal{L}$. Then $\bigvee_{l \in \mathcal{L}} \mathcal{O}_{\alpha,\Gamma,P,T}$ describes lock-order guided formula generation.

Consistency Relaxation

The key idea behind consistency relaxation is to treat each thread in the cluster as an independent entity whose channel inputs are arbitrarily assigned and independent of the outputs of other threads. Since input assignment is arbitrary, the values received by a thread and the values previously sent may not match up, and therefore inter-thread execution may appear to be causally inconsistent. Nevertheless, consistency relaxation is sound in that we know there exists at least one control-plane deterministic run in this ultra-relaxed consistency model—the original run. In particular, the original run is an example of a run that one might obtain when channel inputs are arbitrarily assigned, even though coincidentally the inputs and output of the original run appear to be consistent (by a stroke of great luck).

Figure 5.6 gives the evaluation rules for relaxed consistency formula generation. It differs from the base evaluation rules in two key ways. First, all incoming inputs on data-plane

$$\mathrm{E}_{\alpha,P,T}(t,\Sigma,B) =$$

$$\begin{cases}
\mathrm{E}_{\alpha,P,T}(t,\Sigma[x \mapsto \Sigma(e), pc_t \mapsto_+ 1, i \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := e}, \\
\mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto \alpha_{\Sigma(i+1)}(x)], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsControl}(c) \\
\mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, x \mapsto s_{B(v)}], B[v \mapsto_+ 1]) & \text{if } P_{\Sigma(pc_t)} = \texttt{x := read}_c\texttt{()} \wedge \mathrm{IsData}(c) \\
\Sigma(x) = \alpha_{\Sigma(i)}(x) \wedge \mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsControl}(c), \\
\mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{write}_c\texttt{(x)} \wedge \mathrm{IsData}(c), \\
e \Rightarrow \mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto L, i \mapsto_+ 1, b \mapsto_+ 1], B) \wedge \\
\neg e \Rightarrow \mathrm{E}_{\alpha,P,T}(t,\Sigma[pc_t \mapsto_+ 1, i \mapsto_+ 1, b \mapsto_+ 1], B) & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) < \alpha_n(b), \\
\mathrm{False} & \text{if } P_{\Sigma(pc_t)} = \texttt{if e goto L} \wedge \Sigma(b) = \alpha_n(b), \\
\mathrm{True} & \text{if } P_{\Sigma(pc_t)} = \texttt{exit}.
\end{cases}$$

Figure 5.6: Symbolic evaluation rules for consistency relaxation. Only one arbitrarily chosen thread schedule is explored since data-plane inputs are allowed to be arbitrary and thus are independent of scheduling order.

channels are assigned fresh symbolic variables hence allowing them to be anything regardless of whether the input came from an internal or external node. Second, evaluation is performed along one arbitrarily chosen thread schedule. This is sound as inputs are no longer tied to data that was previously sent on channels.

The key benefit of consistency relaxation is that we need only explore one thread schedule, thus cutting the formula generation complexity by an exponentially-large quantity. The tradeoff however is that the resulting execution, although control-plane deterministic, will likely not correspond to a sequentially consistent execution. In particular, incoming data may be assigned values consistent with the local control plane I/O, but inconsistent with the data that was previously sent on that channel. This lack of consistency may make it difficult to trace causality across nodes.

Chapter 6

Implementation

We implemented DCR for clusters of Linux x86 machines (e.g., such as those available on EC2). DCR consists of approximately 150 KLOC of source code (40% LibVEX and 60% DCR + plugins). Here we demonstrate how to use DCR and then discuss major challenges we faced when implementing it.

6.1 Usage

One can start using DCR in seconds. To record, simply invoke DCR on the application binary, specifying the location to dump log files (e.g., distributed storage) and the location of persistent data files (e.g., an HDFS mount):

```
$ dcr-record --save-as=hdfs://i1/demo
    --persistent-store=/mnt/hdfs/data
    ./mesos-master
```

DCR will then record the application, taking care not to record data-plane inputs originating from the specified persistent storage.

To replay using the DDFLOW analysis plugin (see Section 7.1.1), one need only specify the plugin name and the location of previously collected recordings:

```
$ dcr-replay --plugin=dflow
    hdfs://i1/demo/*
```

6.2 Lightweight Recording

A key observation behind DCR's implementation is that bugs in datacenter applications often originate from within application code rather than from kernel code. After all, datacenter applications rarely involve kernel changes. This observation motivates DCR's approach

of tracing only the non-determinism needed to replay user-level code of developer-selected application processes.

6.2.1 Interpositioning

DCR interposes only on user-level communication channels (sockets/pipes/files and shared-memory) of traced processes.

Sockets/pipes/files are interposed with the help of a DCR's kernel module. The module delivers a signal for every system call invoked by a traced process, which DCR then handles. To address the high syscall rates of some datacenter applications, we also intercept syscalls made through Linux's vsyscall page (a user-level trampoline/layer of indirection into kernel-land), hence avoiding the expense of signals for a majority of syscalls (most libc calls go through the vsyscall).

Shared memory accesses are interposed with the help of DCR's CREW kernel module. The module uses virtual memory page protections to serialize conflicting user-level page accesses.

CREW in detail. Conceptually, DCR's CREW implementation closely follows that of SMP-ReVirt's [17]: it maintains shadow page tables whose permission are upgraded and downgraded at CREW events. But unlike SMP-ReVirt, DCR maintains shadow page tables only for those processes that are traced. Moreover, DCR does not shadow kernel pages (they are identical to those in guest page tables) hence avoiding false sharing in the kernel (a significant bottleneck in SMP-ReVirt). DCR interposes on page table operations using Linux's `paravirt_ops` interface in the same manner as Xen.

6.2.2 Asynchronous Events

A key benefit of replaying at the user-level is that DCR needn't record all interrupts (e.g., device and timers) – something that VM-level replay tools must do in order to ensure kernel code is replayed. The only asynchronous events DCR must record are signals and preemptions.

The key challenge with replaying these events is in ensuring precise delivery of events (i.e., at the same instruction count) during replay. A simple way to do this is to count the number of instructions at record time and deliver the same event at the recorded instruction count in replay. Unfortunately, this method requires the use of a software instruction counter (e.g., implemented via binary translation) and known to incur high runtime overheads.

To address this problem, DCR leverages standard asynchronous even replay technique. These technique rely on the following two observations. First, one can precisely identify a point in program execution via the x86 triple ¡eip, ecx, branch count¿. The second is that

one can efficiently obtain the branch count from the hardware performance counters found in modern commodity machines.

6.2.3 Piggybacking

DCR needs to communicate trace data (logical clocks, unique message ids) to remote nodes during recording, and uses piggy-backing techniques to do so. However, the naive approach of piggy-backing trace data on each network packet results in impractical communication costs.

DCR employs two techniques, both of which leverage the semantics of system calls, to reduce piggy-backing overheads: *message-level piggy-backing* and *TCP-aware unique ids*. The key observation behind message-level piggy-backing is that data plane applications send data in large message chunks: Memcached, for instance, performs `sys_send`s on 2 MB buffers. DCR leverages this observation by piggy-backing at the message level rather than the packet level.

The key observation behind TCP-aware unique ids is that datacenter applications almost always use TCP to transfer data, and that each message in a TCP stream has an implicit unique id within the stream (i.e., it's sequence number). Thus one can obtain a globally unique id for any given TCP message using a ¡stream id, local id¿ tuple. The stream id need only be communicated once when the TCP connection is established, while the local id can be computed during replay based on the ordering of messages received on the stream.

6.3 Group Replay

6.3.1 Serializability

The current implementation of DCR provides the illusion of serial replay by actually replaying nodes serially: only one thread at any given node is allowed to execute at a time. Though simple to implement and verify, the undesirable consequence of this implementation decision is that replay slowdown will increase linearly with the number of thread/nodes being replayed. That is, 1000 nodes will take approximately 1000x as long to replay, even if replay is distributed over 1000 nodes!

We are currently exploring parallelization techniques that offer high concurrency levels while preserving the illusion of serial plugin execution (serializability). In particular, the main challenge is to provide plugin callbacks with a serializable view of distributed state. The current goal is obtain serializability with a simple two-phase locking procedure, using per-page locks. If locking turns out to be a bottleneck, the database literature on concurrency control is rife with other serializability techniques.

6.3.2 Instruction-Level Introspection

DCR plugins have access to a variety of fine-grained analysis primitives such as data-flow tracking and instruction tracing. Under the hood, DCR implements these primitives by binary translating the replay execution. The binary translation is done by LibVEX, an open-source binary translator that offers an easy-to-use RISC-style intermediate representation for performing instruction-level analyses. Each analysis primitive, then, is implemented as a LibVEX analysis module,

A key challenge in binary translated replay is that LibVEX is neither complete nor precise. It is incomplete in that, it does not simulate operations on hardware performance counters. This is a problem because we rely on performance counters to tell us when to deliver asynchronous events during replay. DCR addresses this problem by adding branch counting emulation support to LibVEX (in the form of a module that counts branches in software).

LibVEX is imprecise in that, even though it supports FPU emulation, it can only do so with 56-bit precision. Thus the results of FPU computation done in binary translation may not be the same as those 64-bit FPU computations done in record-mode (i.e., under direct execution), often causing replay-mode divergence. We workaround this problem, with some penalty, by binary translating FPU operations (and only FPU operations) even during record mode. Luckily, datacenter applications are usually not FPU intensive. To detect FPU operations, we set the appropriate bits in the x86 control register, hence causing subsequent FPU operations to trap.

Chapter 7

Evaluation

We evaluate DCR using two metrics: its effectiveness as a debugging aid (Section 7.1) and its performance in terms of record and replay mode overheads (Section 7.2). In a nutshell, we found that DCR is effective as a platform for the construction of powerful debugging tools, and for assisting in the debugging of hard distributed bugs. Moreover, we found that although DCR exhibits long replay times due to inference, it has low record-mode overheads.

7.1 Effectiveness

We gauge DCR's effectiveness with two qualitative metrics. The first, analysis power, measures the ease by which developers can construct sohphisticated analysis plugins using the DCR infrastructure. To this end, Section 7.1.1 focuses on powerful distributed data flow analysis we built using the DCR framework. The second metric, debugging experience, measures the ease by which datacenter developers can debug real world bugs. Section 7.1.2 presents several debugging case studies toward that end.

7.1.1 Analysis Power

We've built several powerful analysis plugins on top of DCR. Here we focus on our most powerful plugin–distributed data flow analysis.

Distributed Data Flow

Here we focus on the design of DDFLOW—a distributed data-flow analysis plugin we've built using the DCR toolkit. DDFLOW's goal is to provide a trace of all instructions or functions that operate, transitively, on the contents of a (user-specified) origin data file or message. DDFLOW is particularly useful in the context of data-loss bugs: it lets you track the flow of your data and helps you quickly identify where your data finally ends up – a process that could take hours of manual searching if done manually. DDFLOW highlights the power of DCR plugins

because it is an example of a heavyweight analysis that can most easily be done during replay (in-production datacenter apps will timeout due to the overhead of such heavyweight analysis).

The DDFLOW Python plugin can be given in just a few lines (initialization code is committed):

```
msg_taint_map = {}
def on_send(msg):
   if msg.is_tainted():
      msg_taint_map[msg.id] = 1
def on_recv(msg):
   if msg_taint_map[msg.id]:
      local.set_taint(msg.rcvbuf)
   else:
      local.untaint(msg.rcvbuf)
   del msg_taint_map[msg.id]
```

DDFLOW works by propagating taint within and across nodes. To track taint within, DDFLOW relies on DCR's data-flow primitive. To track taint across nodes, DDFLOW maintains a Python map of tainted messages (updated via the on_send and on_recv callbacks), keyed on unique message IDs (provided by DCR). When a tainted message is received, DDFLOW updates taint state for the receiving buffer.

7.1.2 Debugging

In this section we describe how we used DCR to succesfully reproduce and debug bugs in Hypertable [2]–an open source high performance data storage designed for large-scale data-intensive tasks and is modeled after Google's Bigtable [12]. Hypertable is deployed at Baidu, the leading search service in China and the Rediff online news provider.

Data Loss in Hypertable

We used DCR to debug a previously-solved Hypertable defect [3] that causes updates to a database table to be lost when multiple Hypertable clients concurrently load rows into the same table. This bug is hard to reproduce and its root cause spans across multiple nodes. The load operation appears to be a success—neither clients nor slaves receiving the updates produce error messages. However, subsequent dumps of the table do not return all rows—several thousand are missing. The data loss results from rows being committed to slave nodes (i.e., Hypertable range servers) that are not responsible for hosting them. The slaves honor subsequent requests for table dumps, but do not include the mistakenly committed rows in the dumped data. The committed rows are merely ignored. The erroneous commits stem from a race condition in which row ranges migrate to other slave nodes at the same time

that a recently received row within the migrated range is being committed to the current slave node.

Reproducing this failure required 8 concurrent clients that insert 500MB data into the same table, after which they check the consistency of the table. We recorded several executions with DCR until the failure was reproduced—the recording overhead was similar to the one in Figure 7.10. Afterwards, we replayed the failure with DCR in our development single-machine setup. We inserted breakpoints during row range migration, where we suspected the root-cause is located and we observed the data race occurring deterministically. DCR's ability to reliably replay the failure combined with the bird's eye view of the entire system made debugging substantially easier and faster.

Hypertable Hang Under Memory Pressure

We accidentally discovered a new bug in Hypertable while recording various workloads with DCR. We noticed that occasionally Hypertable clients timed-out and the system became unresponsive. This failure was hard to reproduce without DCR, so we recorded subsequent executions with DCR until the error manifested again. It turned out that the error would manifest when the machine where the Hypertable master server was running experienced memory pressure and a memory allocation failed, which in turn hanged the master. Once it recorded the failed execution, DCR's deterministic replay and the visualization plugin helped to quickly identify that nodes were trying to connect to the master, which was not making any progress. We identified the failed memory allocation, which explained the random Hypertable hangs that we were experiencing. On subsequent analysis, we discovered that particular cluster machine was accidentally configured without a swap partition, making memory allocations more likely to fail.

7.2 Performance

DCR's record and replay performance depends on the particular configuration of DCR that is used, and therefore, we evaluate DCR's performance under two different configurations. The first configuration, termed DCR-SHM is geared toward providing efficient recording and replay for shared-memory intensive multi-processor applications. Toward this goal, all DCR-traced nodes record all inputs and outputs (control and data), a lock ordering (a type of partial ordering), and a sample of the path taken by each CPU. It then uses DRI, along with input, path, and lock-order guidance optimizations to compute a replay run. DCR-SHM incurs high record-mode penalties for I/O intensive workloads (since it records inputs and outputs). However, because DCR-SHM records only the lock ordering of CPUs, it incurs little overhead for CPU-intensive datacenter appliations, including those with multiprocessor data-races.

The second configuration, which we term DCR-IO, is geared toward providing efficient recording and replay for I/O intensive (specifically, disk and network intensive) apps. In

this configuration, all DCR-traced nodes record control-plane I/O, and the order of channel communication (both shared-memory and network). In addition, boundary nodes record pointers to persistent data-plane inputs. With this information, DCR-IO then employs DRI along with the input and lock-order guidance optimizations to compute a replay run. Recording the order of channel communication enables DCR-IO to speed inference considerably (in fact, it eliminates the need for any inference), but incurs high record-mode overheads for shared-memory intensive multiprocessor applications, particularly those that exhibit false-sharing.

7.2.1 DCR-SHM

We evaluate DCR-SHM under two configurations: (1) DCR-SHM-PARTIAL in which DCR-SHM records only some of the branch outcomes during the original run, and (2) DCR-SHM-TOTAL in which DCR-SHM records all branch outcomes (i.e., all CPU paths). We begin with our experimental setup and then give results for each major DCR phase: record, inference, and replay. In summary, we found that when using DCR-SHM-PARTIAL, DCR incurs low recording overhead (less than 1.6x on average), but impractically high inference times. For instance, two applications in our suite took more than 24 hours during the inference phase. In contrast, DCR with DCR-SHM-TOTAL incurs significantly higher recording overhead (between 3.5x and 5.5x slowdown of the original run). But the inference phase finished within 24 hours for all applications. Thus, DCR-SHM-PARTIAL and DCR-SHM-TOTAL represent opposite design points in the record-inference tradeoff space.

Setup

We evaluate seven parallel applications: *radix*, *lu*, and *water-spatial* from the Splash2 suite [48], the Apache web-server (*apache*), the Mysql database server (*mysql*), the Hotspot Java Virtual Machine running the Tomcat webserver (*java*), and a parallel build of the Linux kernel (*make-j2*). We do not give results for the entire Splash2 suite because some (e.g, FMM) generate floating point constraints, which our current implementation does not support. Henceforth, we refer to the Splash2 applications as *SP2 apps* and the others as *systems apps*.

All inputs were selected such that the program ran for just 2 seconds. This ensured that inference experiments were timely. *Apache* and *java* were run with a web-crawler that downloads files at 100KBps using 8 concurrent client connections. *Mysql* was run with a client that queries at 100KBps, also using 8 concurrent client connections. The Linux build was performed with two concurrent jobs (*make-j2*).

Our experimental procedure consisted of a warmup run followed by 6 trials. We report the average numbers of these 6 trials. The standard deviation of the trials was within three percent. All experiments were conducted on a two-core Dell Dimension workstation with a

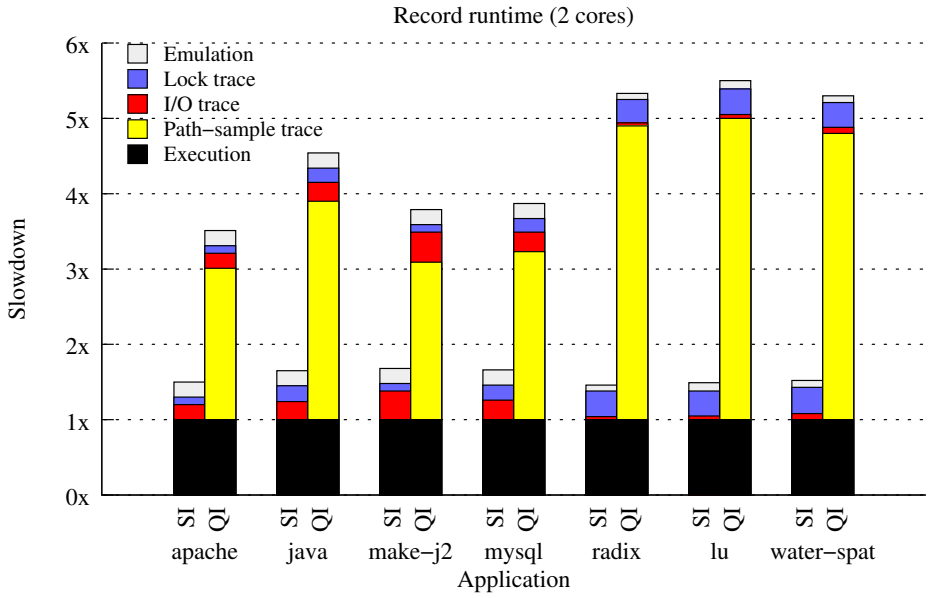

Figure 7.1: DCR's record-mode runtimes, normalized with native application execution time, for both DCR-SHM-PARTIAL (first bar) and DCR-SHM-TOTAL (second bar).

Pentium D processor running at 2.0GHz and 2GB of RAM. The OS used was Debian 5 with a 2.6.29 Linux kernel with minor patches to support DCR's interpositioning hooks.

DCR-SHM-PARTIAL Record Mode

Figure 7.1 shows the record-mode slowdowns when using DCR-SHM-PARTIAL. The slowdown is broken down into five parts: (1) *Execution*, the cost of executing the application without any tracing or interpositioning; (2) *Path-sample trace*, the cost of intercepting and writing path-samples to the log file; (3) *I/O trace*, the cost of intercepting and writing both program input and output to a log file; (4) *Lock trace*, the cost of intercepting bus-lock instructions and writing logical clocks to the log file at each such instruction; (5) *Emulation*, the cost of emulating some syscalls.

As shown in Figure 7.1, the record mode causes a slowdown of 1.6x on average. DCR outperforms software-only multiprocessor replay systems on key benchmarks, and is comparable on several others. For instance, DCR outperforms SMP-Revirt [17] on *make-j2* (by 0.3x) and *radix* (by 0.1x) for the 2-processor case. DCR does better because these apps exhibit lots of false-sharing. False-sharing induces frequent CREW faults on SMP-ReVirt, but not on DCR since it does not record races. DCR approaches RecPlay's [44] performance (within 0.4x) for the SP2 apps. This is because, with the exception of outputs and sample points, DCR traces roughly the same data as RecPlay (though RecPlay captures lock order at the library level). SP2 apps are not I/O intensive, so the fact that DCR records the outputs does not have a

significant impact.

DCR does not always outperform existing multiprocessor replay systems. For instance, in the two-processor case, SMP-ReVirt and RecPlay achieve near-native performance on several SP2 apps (e.g., LU), while DCR incurs an average overhead of 0.5x of SP2 apps. As Figure 7.1 shows, a bulk of this overhead is due to lock-tracing, which SMP-ReVirt does not do. And while RecPlay does trace lock order, it does so by instrumenting lock routines (in `libpthreads`) rather than all locked instructions. Intercepting lock order at the instruction level is particularly damaging for SP2 app performance because they frequently invoke `libpthreads` routines, which in turn issue many locked-instructions. One might expect the cost of instruction-level lock tracing to be even higher, but in practice it is small because `libpthread` routines do not busy-wait under lock contention – they await notification in the kernel via a `sys_futex`. Nevertheless, these results suggest that library-level lock tracing (as done in RecPlay) might provide better results.

Compared with hardware-based systems, DCR performs slightly worse, especially on systems benchmarks. For example, CapoOne [39] achieves a 1.1x and 1.6x slowdown for *apache* and *make-j2*, respectively, while DCR achieves a 1.6x and 1.8x slowdown. Based on the breakdown in Figure 7.1, we attribute this slowdown to two bottlenecks. The first, not surprisingly, is output-tracing. The effect of output tracing is particularly visible in the case of *apache* and *make-j2*, which transfer large volumes of data. The second bottleneck is the emulation. Triggering a signal on each syscall and emulating task and memory management at user-level can be costly.

DCR-SHM-PARTIAL Inference Mode

Figure 7.2 gives inference slowdown for each application. The slowdown is broken down into 4 major DCR-SHM-PARTIAL stages: schedule-select, input-select, path-select, and formula generation and solving (*FormGen+FormSolve*). The path-select and *FormGen+FormSolve* stages account for a vast majority of the inference time. Since its query does not contain a path or read-trace, DCR-SHM-PARTIAL has to search for them. In contrast, schedule-select and input-select are instantaneous due to consistency relaxation (Section 5.2.2) and guided search (Section 5.2.1), respectively.

Overall, DCR-SHM-PARTIAL's inference time is impractically long. Two applications, *java* and *mysql*, do not converge within the 24 hour timeout period, and those that do converge achieve an average slowdown of $12,232$x. As the breakdown in Figure 7.2 shows, there are two bottlenecks. The primary bottleneck is path-selection, taking up an average 75% percent of inference time. In the case of *java* and *mysql*, path-selection takes so long that it prevents DCR from proceeding to the *FormGen+FormSolve* within the timeout period (24 hours). The secondary bottleneck is *FormGen+FormSolve*, taking up the remaining average 25% percent of inference time. We investigate each bottleneck in the following sections.

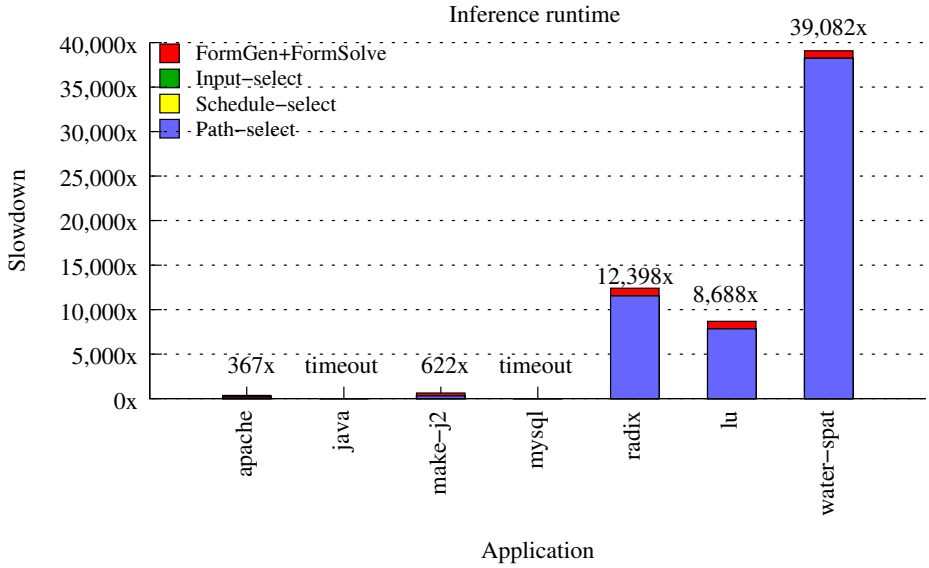

Figure 7.2: Inference runtime, normalized with native application execution time, for `DCR-SHM-PARTIAL`. Applications that did not finish in the first 24 hours are denoted by *timeout*.

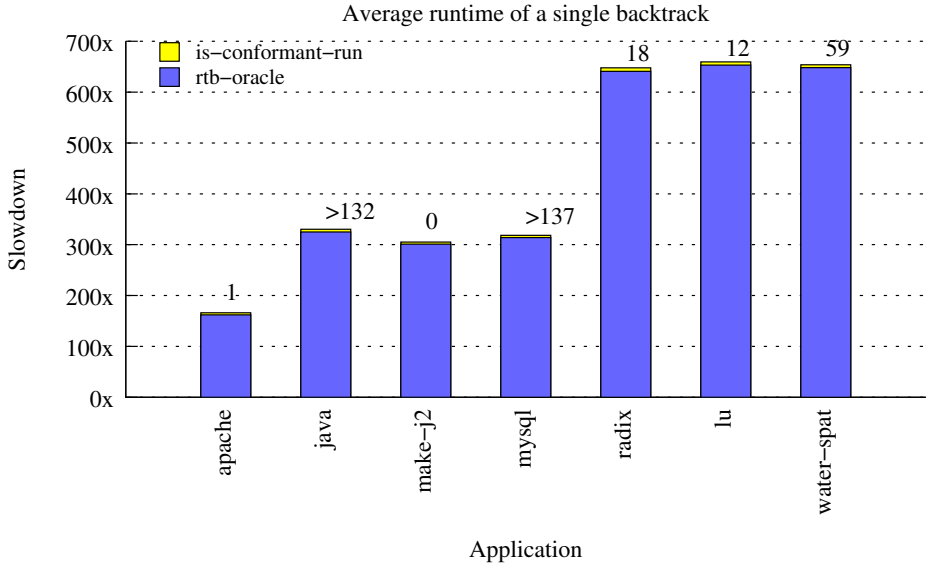

Figure 7.3: The average runtime, normalized with native runtime, of one backtrack performed by `DCR-SHM-PARTIAL` during PATH-SELECT, broken down into its two subroutines. The total number of backtracks is given at the top of each bar. For applications that timeout, this number is just a lower-bound.

Path Selection. We expected path selection to be the primary cause of `DCR-SHM-PARTIAL`'s slowdown. After all, `DCR-SHM-PARTIAL`'s path selection algorithm (PATH-SELECT) may backtrack an exponential number of times. We also expected the cost of each backtrack to play a strong secondary role. To verify these expectations, we counted the number of backtracks and measured the average cost of a backtracking operation. The results, shown in Figure 7.3, contradict our expectations. That is, the number of backtracks for most apps, with the exception of *java* and *mysql*, is low, hence making the cost of each backtrack operation the dominant factor.

There are two reasons for the small number of backtracks. The first, specific to *make-j2* and *apache*, is that there are a small number of dynamic races and consequently a small number of race-tainted branches (RTBs). *make-j2*, for instance, does not have any shared-memory races at user-level, and hence no divergences that induce backtracks. Most sharing in *make-j2* is done via file-system syscalls, the results of which we log, rather than through user-level shared memory. *Apache*, in contrast, does have races and RTBs, but a very small number of them. Our runs had between 1 and 2 dynamic races, each of which tainted only 1 branch. Thus, in the worst case, we would have to backtrack 4 times. The actual number of backtracks is smaller because PATH-SELECT guesses some of these RTBs correctly on the first try.

The second reason for the small number of backtracks is specific to the SP2 apps. These apps did well despite having a large number of RTBs (an average of 30) because PATH-SELECT was able to resolve all their divergences with just one backtrack, hence making forward-progress without exponential search. Only one backtrack was necessary because, in the code paths taken in our runs, there is a sampling point after every RTB. So if PATH-SELECT chooses an RTB outcome that deviates from the original path, then the following sampling point will be missed, hence triggering an immediate backtrack to the original path.

Unlike the majority of apps in our suite, *java* and *mysql* have a significantly larger number of backtracks. The large number stems from code fragments with few sampling points between RTBs. In those cases, we end up with too many instructions between successive sampling points, hence resulting in divergences that require a large number of backtracks to resolve. Consider a loop that contains a race, and no sampling points. Thus, the earliest point we can detect a divergence is at the first sampling point after the loop finishes. Now assume that the loop executes 1,000 times, but the divergence is caused at the 500-th iteration. In this case, we need to backtrack 500 times to identify the cause of the divergence.

Overall, our results indicate that reducing the backtracking cost is key to attaining practical inference times – 1000 backtracks may be tolerable if each incurs say at most a 2x slowdown. To identify opportunities for improvement, we broke down the average backtracking slowdown into its two major parts, shown in Figure 7.3. The first part is the cost of invoking the RTB-ORACLE, needed to intelligently identify backtracking points. The second is the cost of invoking IS-CONFORMANT-RUN, needed to verify that a selected path is path-template conformant. The results show that the cost of a backtrack is dominated by the invocation of the RTB-ORACLE, as expected. We expected the RTB-ORACLE to be expensive

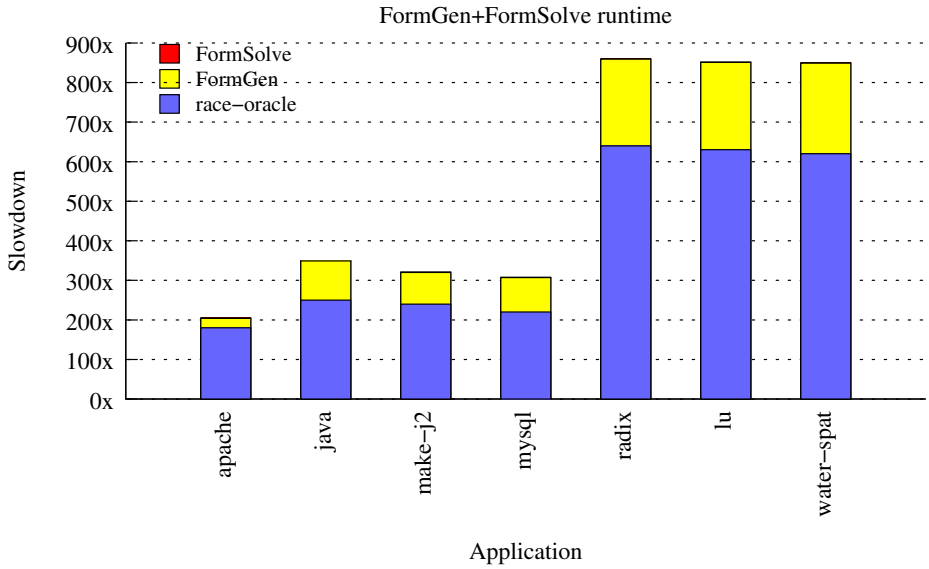

Figure 7.4: Runtime of the *FormGen* and *FormSolve* phases, normalized with native runtime, broken down into its three stages.

because it involves race detection and taint-flow analyses over the entire path up till the point of divergence. In theory, the RTB-ORACLE need not be run over the entire failed path on every backtrack, but we leave that to future work.

Formula Generation and Solving. We expected formula generation (*FormGen*) and solving (*FormSolve*) to be slow, especially since *FormSolve* is an NP-complete procedure for worst-case computations (e.g., hash-functions). To verify this hypothesis, we broke down the phase's runtime into three parts, as show in Figure 7.4. The first part is the cost of invoking the race-oracle, needed to identify which accesses may race. The second part is *FormGen*, used to encode a set of candidate read-traces as a logic formula. And the third part is *FormSolve*, needed to find an output-deterministic read-trace from the candidate set, which in turn involves invoking a formula solver. The breakdown contradicted our hypothesis in that most of the inference is spent in the race-oracle (a polynomial time procedure), not *FormGen* or *FormSolve*.

FormGen and *FormSolve* are fast for two reasons. The first is that our formula generator (Chapter 5) generates formulas only for those instructions influenced (i.e., tainted) by racing accesses. If the number of influenced instructions is small, then the resulting formula will be small. Our results indicate that, for the apps in our suite, races have limited influence on instructions executed – the average size of a formula is 1562 constraints. The second reason is that, of those constraints that were generated, all involved only linear twos-complement

arithmetic. Such constraints are desirable because the constraint solver we use can solve them in polynomial time [23]. We did not encounter any races that influenced the inputs of hash functions or other hard non-linear computations.

The penalty for efficient constraint generation and solving is expensive race-detection. Our race-oracle is slow because it performs set-intersection of all accesses made is a given path. Because a set may contain millions of accesses, the process is inherently slow, even with our $O(n)$ set-intersection algorithm.

`DCR-SHM-TOTAL` Record and Inference Modes

As we have shown so far, `DCR-SHM-PARTIAL` leads to a low recording overhead, but its inference time is prohibitive. In this section, we evaluate `DCR-SHM-TOTAL`, which trades recording overhead for improved inference times. In particular, `DCR-SHM-TOTAL` relies on recording branches during the original run, as explained in Section 4.2.1. Recording branches removes the need to invoke PATH-SELECT, the key bottleneck behind the timeouts in `DCR-SHM-PARTIAL`. Hence the improvements in the inference time.

Figure 7.1 shows `DCR-SHM-TOTAL`'s slowdown factors for recording, normalized with native execution times. As expected, recording branches significantly increases the overhead, from 1.6x to 4.5x on average. While this overhead is still 4 times less than the average overhead of iDNA [10], it is greater than other software-only approaches, for some apps. *R*adix, for example, takes 3 times longer to record on `DCR` when using `DCR-SHM-TOTAL` than with SMP-ReVirt [17].

Figure 7.5 shows the inference time for `DCR-SHM-TOTAL` normalized with native execution times. As expected, `DCR-SHM-TOTAL` achieves much lower inference times than `DCR-SHM-PARTIAL` (see Figure 7.2). The improvements are due to the fact that `DCR-SHM-TOTAL` does not need to spend time in the path-select sub-stage of read-trace guidance (the most expensive part of `DCR-SHM-PARTIAL`) since the original path of each thread has been already recorded. In the absence of path-selection overhead, the *FormGen+FormSolve* stage dominates. As illustrated in Figure 7.4, the largest percentage of time in the *FormGen+FormSolve* stage is spent in the race-oracle.

Replay Mode

Figure 7.6 shows replay runtime, normalized with native runtime, broken down into the costs of replaying various types of non-determinism.

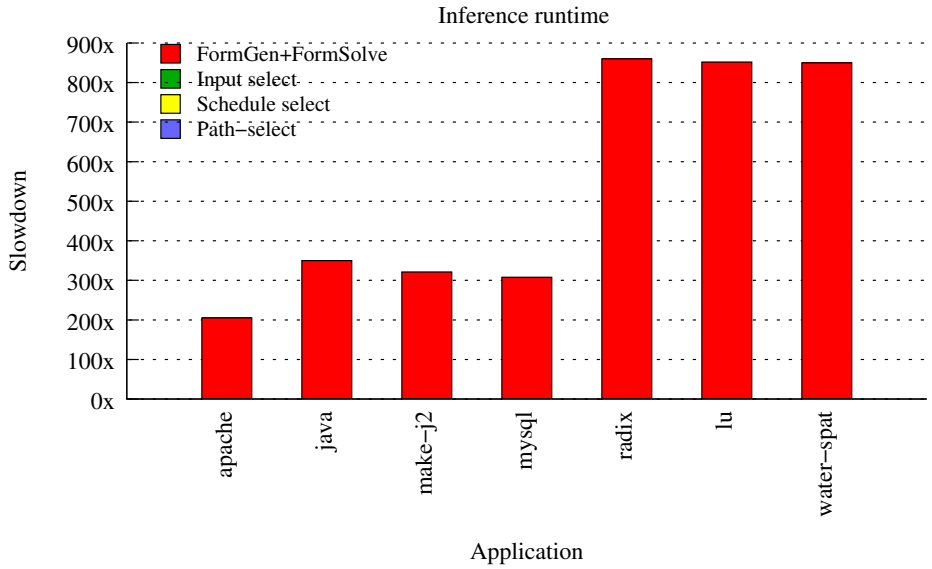

Figure 7.5: Inference runtime overheads for `DCR-SHM-TOTAL`. All applications finish well within the first 24 hours.

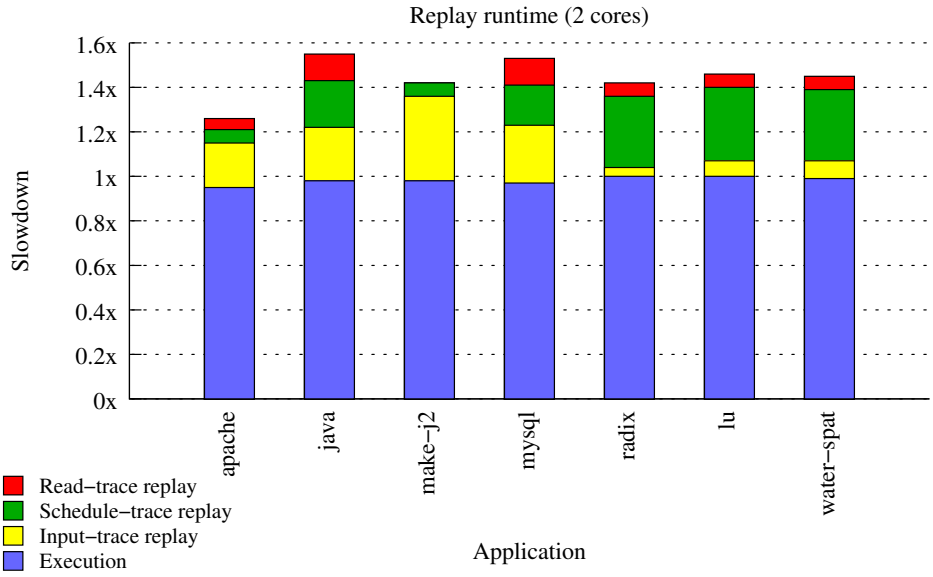

Figure 7.6: Two-processor replay-mode runtime overheads normalized with native execution time.

The surprising result here is that schedule-trace replay and read-trace replay are much faster than what one might expect of a serialized execution in which all reads are intercepted. Two additional optimizations employed by DCR explain these results. For the first optimization, rather than follow the schedule-trace precisely, DCR simply replays the lock-order of the schedule-trace (which for DCR-SHM-PARTIAL and DCR-SHM-TOTAL is the original lock-order). For the second optimization, rather than replay all read-values in the read-trace, DCR replays just those of racing reads.

Replay speed is not near-native largely due to the cost of intercepting and replaying lock-instructions, a key bottleneck in record mode as well. Applications with a high locking rate (e.g., *java*, SP2 apps) suffer the most. We hope to improve these costs in a future implementation, perhaps by moving to a library-level lock interception scheme. As with other replay systems, native execution time in replay mode is smaller for I/O intensive apps because DCR skips the time originally spent waiting for I/O.

7.2.2 DCR-IO

We evaluate DCR-IO on real-world I/O intensive datacenter applications. We begin with our experimental setup and then give results for record and replay modes. In short, we found that DCR-IO's record overheads are low (at about 1.3x) and scale well to massive datasets, both in uniprocessor and multiprocessor modes. Morover, we found that DCR-IO's replay times are orders-of-magnitude faster than DCR-SHM's since it does not require inference (recall that it records control plane I/O and inter-CPU communication ordering, and reuses persistent data-plane inputs).

Setup

We evaluate DCR-IO on two major datacenter applications: *Hypertable* [2] and *memcahed* [4]. Hypertable is an open source, high performance data storage designed for large-scale data-intensive tasks and is modeled after Google's Bigtable [12]. It is deployed at Baidu, the leading search service in China and the Rediff online news provider. Memcached is an open source, high performance distributed object caching system designed to speed data retrieval. It is used by many companies, including Google and Facebook.

We ran all experiments in a cluster with 14 machines with 2 Intel Xeon 3.06GHz processors, 2GB of RAM, two 7200RPM drives in RAID 0, running 32-bit Linux 2.6.29. The machines are in a single rack, have 1Gbps NICs, and are interconnected by a single 1Gbps switch. The size of the cluster may not be representative of the size of current data centers, however, we used the largest cluster that was available to us and in which we had access to the bare-metal hardware. We could not use a virtualized environment such as EC2 because we needed access to the hardware branch counter in order to replay asynchronous events.

Record Mode

We measured DCR's recording performance versus the slowdown of the naive approach that records all inputs, in order to show the benefits of REPLAYNET. To simulate the naive approach, we configured DCR to log all inputs. We first evaluate the single processor case, then the logging overhead (§7.2.2) and then the multiple CPU case.

Runtime Overhead. We first evaluate the single processor case, therefore the CREW protocol was not used. To use a single CPU, we set the CPU affinity to a single CPU for both the native and the recorded systems.

Memcached. Memcached [4] is a high-performance, distributed memory object caching system, typically used for speeding dynamic web applications by alleviating database load. Memcached is used by online services providers such as Youtube, Wikipedia, Flickr, etc.

To evaluate the efficiency of recording a memcached deployment, we simulated a photography blog Web application in which memcached is used by the user-facing Web application server to cache the files containing the photos. This setup resembles the Facebook photo storage [8], in which memcached is used to reduce latency. We assume that the photos are stored in persistent storage (i.e., HDFS) and the clients copy them from persistent storage to the memcached servers. We used various setups with a varying number of memcached servers, number of clients, and total input sizes. Each server and client run on separate machines. Each client randomly selects one of the memcached servers to either write or read a photo—reads are selected with 90% probability since reads are predominant in Facebook's daily photo traffic [8].

DCR's recording overhead with varying size of the input from persistent storage (Figure 7.7) is between 18% and 23%. On the other hand, the naive approach imposes a high overhead: between 100% and 125%. This shows the benefits of REPLAYNET: logging all inputs causes the naive approach to have up to 5 times higher runtime overhead than DCR.

For this experiment we used a setup consisting of 4 memcached servers and 7 clients, each client having 4 threads. Slowdown is measured in terms of reduction in client throughput. In the baseline execution, clients achieve a maximum throughput of 68MB/s, corresponding to 68 memcached operations per second. The photos were configured to have a fixed size of 1MB, they were randomly generated and previously stored in the clients' local disks before starting the experiment.

Figure 7.8 shows DCR's scalability with the number of nodes in the system. We varied the number of recorded nodes by increasing the number of memcached clients. Each client connects to a shared pool of 4 memcached servers. The slowdown is measured in terms of reduction in client throughput.

This experiment shows that DCR's overhead is between 20% and 65% and scales well with the number of nodes in the system. Moreover, DCR scales well when the servers operate under heavy load. The naive approach has high overhead (up to 250%). However, as the mem-

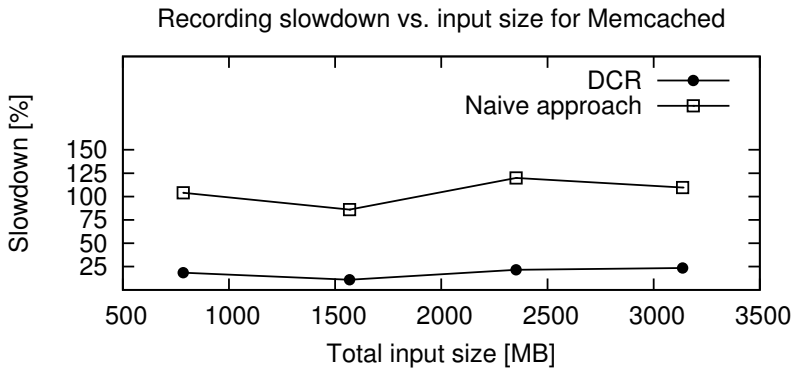

Figure 7.7: Recording slowdown in Memcached while varying the total size of the input from persistent storage.

cached servers become saturated, clients become less loaded. Since in the naive approach clients are responsible for most of the logging (the workload is dominated by reads, which are fully recorded by clients), the impact of heavy logging for the naive approach decreases.

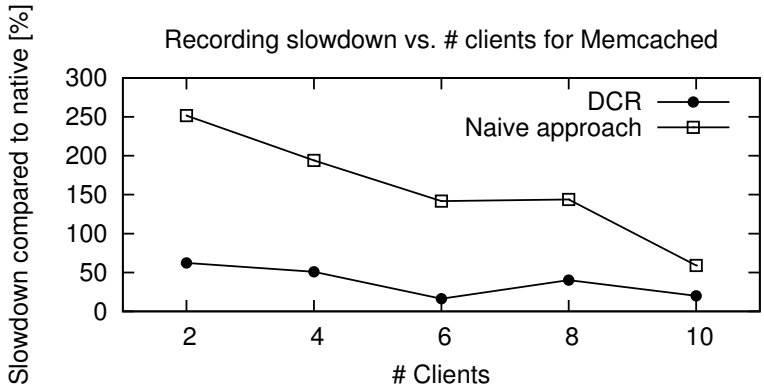

Figure 7.8: Recording slowdown for Memcached with varying number of clients.

Hypertable. The Hypertable workload consists of several clients inserting a log of Web search queries and click streams into a Hypertable table. A query is several hundred bytes long and contains the timestamp, user id, the query keywords and the links clicked by the user. The clients perform a workload that would be performed by a user-facing component of the data center, such as a Web application server.

The range servers store the content of the database tables in memory and also dump them to a distributed file system such as HDFS. Because `DCR` currently requires that the

target application uses a VFS-like interface to communicate with the file system (§4.2.2) we used a dedicated machine in our cluster as a dedicated shared file system for the range servers. In future work we intend to use the HDFS Fuse support and fix a bug in Hypertable that prevented us from experimenting with this setup.

Figure 7.9 shows that for Hypertable, the recording overhead scales well with the size of the input from persistent storage. The overhead, measured as transaction throughput, is in between 10% and 50%. On the other hand, the naive approach has higher overhead, which increases up to 90% for the largest total input size. In this experiment, Hypertable was configured with one master, one lock server, 3 range servers, and 7 clients that placed a heavy load on the system. Each client used an input file ranging from 30MB to 150MB. Clients read the input file from persistent storage.

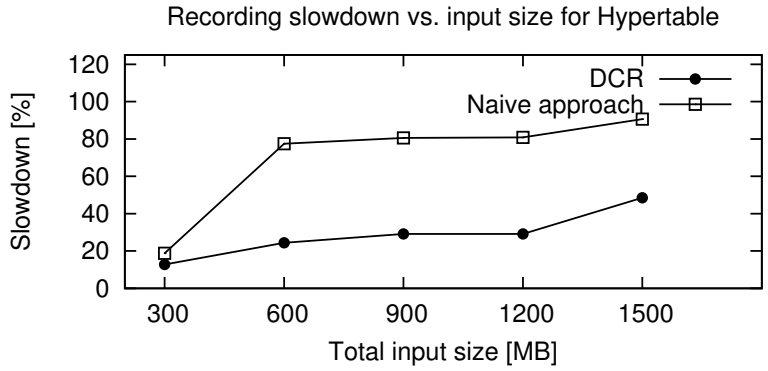

Figure 7.9: Recording slowdown for Hypertable with varying size of the input from persistent storage.

Figure 7.10 shows that DCR scales well with the number of recorded nodes and the overhead is in between 40% and 50%. Due to higher logging rates, the naive approach has higher overhead. In this experiment, Hypertable was configured with one master, one lock server, 2 range servers, and a number of clients ranging from 3 to 9. Each component was run on a separate machine. The overhead is measured in terms of throughput loss.

Log Size. DCR has low logging rates. Figure 7.7 shows DCR's log size for a memcached workload, while varying the total input size read from persistent storage. The naive approach also records internal inputs, therefore it produces an order of magnitude larger logs. For both systems the log size increases linearly with the input size, yet the slope is larger for the naive approach. Moreover, memcached is designed so that server instances do not communicate with each other. If this would have been the case, we would expect that the log size for the naive approach to increase even more, due to the communication between memcached servers, while DCR does not record this communication.

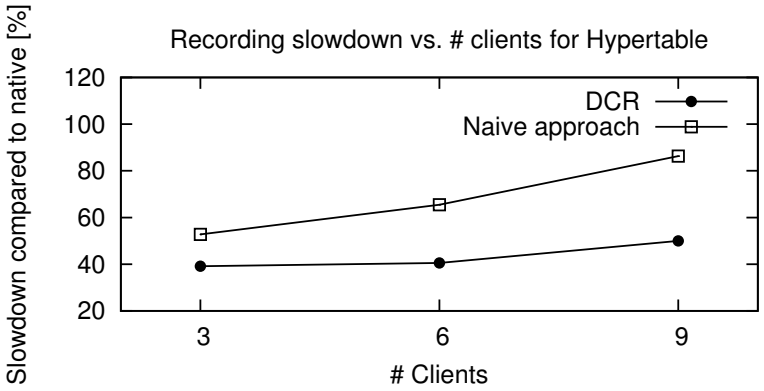

Figure 7.10: Recording slowdown in Hypertable with varying number of clients.

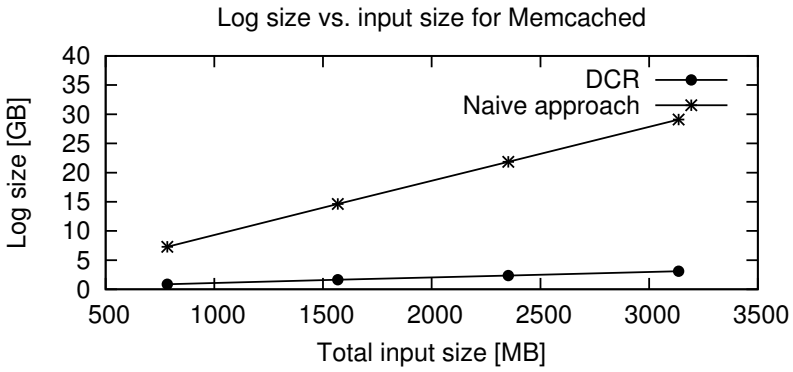

Figure 7.11: Log size for recording a memcached workload with varying input size from persistent storage. DCR has 10 times smaller logs compared to the naive approach.

Hypertable exhibits a similar behavior (Figure 7.12). We expect these results to improve even more with a simple optimization: our current REPLAYNET prototype allocates a static 15KB entry for recording the meta-data associated with an I/O system call. However, this is typically too large: for Hypertable, log entries are dominated by zeros, which we could compress to 100X smaller size. By adding support for variable entry sizes, we expect DCR's logging rates to improve substantially.

Performance for Multi-Processors

To validate our assumption about applying CREW selectively to control plane components such as the Hypertable master and lock server, we enabled CREW in the experiment in Figure 7.10. Thus, the control plane components were allowed to take advantage of both CPUs of their machines. In both cases, DCR had the same slowdown compared to the baseline,

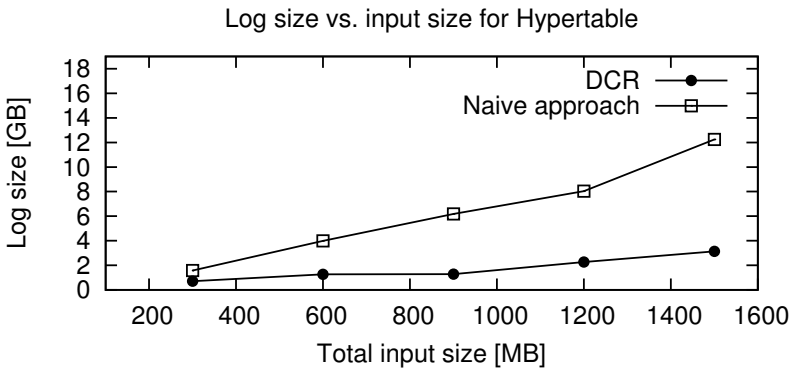

Figure 7.12: Log size for recording a Hypertable workload while varying the input size from persistent storage.

showing that using CREW for the control plane components in Hypertable does not slow down the execution even when the system is under heavy load. This is confirmed by the small rate of CREW faults (at most 150 faults / sec) for each of these components.

To validate the assumption that CREW imposes a high overhead for the data plane components of the system, we recorded also the Hypertable data plane components (the range servers) using CREW and observed overheads larger than 400%.

These experiments confirm our assumptions that turning on CREW for the control plane components is likely to impose low overhead, while having CREW turned on all the time for data plane components is not practical for production use. However, this assumption may not hold for all data center applications and we are in the process of evaluating this further.

Inference Mode

Replay is serial, therefore replay slowdown is expected to be proportional with the number of recorded nodes. For replaying a memcached workload similar to the one in the previous experiments with 3 nodes (one server and 2 clients) replay was 2.46 $\times$ slower than the original run. We also replayed a Hypertable workload similar to the previous experiments. The Hypertable setup consisted of a lock server, a master server, two range servers, and 3 clients. The replay was 2.7 $\times$ slower than the original run. Both experiments were done for the case when all inputs were recorded due to a small bug that made prevented us from using `REPLAYNET`. However, we observed that typically for these applications, replay slowdown using `REPLAYNET` is similar to the replay slowdown of the naive approach. In both these experiments the replay was not n times slower, where n is the total number of nodes. This is because replay can fast forward some operations by eliminating "dead cycles". For instance, operations such as sleep or blocking I/O can complete faster during replay. In real setups, such dead cycles may also arise from multiple applications sharing the same node.

To verify that replay is correct, we also recorded each workload using a debug build of DCR, which also records the inputs and outputs of each recorded node. During replay, it checks that the replayed nodes produce the same outputs as the ones that were recorded and that the branch counter of each system call is the same as during recording.

Chapter 8

Future Work

To our knowledge, DCR is the first system capable of providing low record overhead deterministic replay for existing large-scale, data-intensive distributed applications running on clusters of commodity multiprocessors. Nevertheless, deterministic replay is far from a solved problem, and much work remains to be done before it is truly practical. This chapter focuses on DCR's limitations and potential ways of overcoming them.

8.1 Beyond Control-Plane Determinism

DCR advocates and relies on control-plane determinism—the notion that it suffices to reproduce the behavior of the control plane—the most error-prone component of the datacenter application. In this dissertation, we've argued that control-plane determinism is the ideal for a narrow band of applications ("datacenter applications"). However, two key issues remain unaddressed. First, it is far from clear what the ideal determinism model for arbitrary applications might be. And second, how might we tell whether control-plane determinism comes close to the ideal determinism model for arbitrary applications?

Section 8.1.1 argues that the ideal determinism model of replay debugging is **debug determinism**. The key observation behind debug determinism is that, to provide effective debugging, it suffices to reproduce *some* execution with the *same failure* and the *same root cause* as the original. A debug-deterministic replay system enables a developer to backtrack from the original failure to its root cause. In Section 8.1.2 we propose a metric by which the utility of a determinism model may be assessed with respect to the ideal of debug determinism.

8.1.1 Debug Determinism

We argue that the ideal replay debugging system should provide *debug determinism*. Intuitively, a debug-deterministic replay system produces an execution that manifests the same

failure and the same *root cause* (of the failure) as the original execution, hence making it possible to debug the application. The key challenge in understanding debug determinism is understanding exactly what is a failure and what is a root cause:

A **failure** occurs when a program produces incorrect output according to an I/O specification. The output includes all observable behavior, including performance characteristics. Along the execution that leads to failure, there are one or more points where the developer can fix the program so that it produces correct output. Assuming such a fix, let P be the predicate on the program state that constrains the execution—according to the fix—to produce correct output. The **root cause** is the negation of predicate P.

A *perfect implementation* fully satisfies the I/O specification, that is, for any input and execution it generates the correct output. A deviation from the perfect implementation may lead to a failure. So, more intuitively, this deviation represents the root cause.

In identifying the root cause, a key aspect is the boundary of the system: e.g., if the root cause is in an external library (i.e., the developer has no access to the code), a fix requires replacing the library. Else, if the library is part of the system, the fix is a direct code change.

Debug determinism is the property of a replay-debugging system that it consistently reproduces an execution that exhibits the same root cause and the same failure as the original execution.

For example, to fix a buffer overflow that crashes the program, a developer may add a check on the input size and prevent the program from copying the input into the buffer if it exceeds the buffer's length. This check is the predicate associated with the fix. Not performing this check before doing the copy represents a deviation from the ideal perfect implementation, therefore this is the root cause of the crash. A debug-deterministic system replays an execution that contains the crash and in which the crash is caused by the same root cause, instead of some other possible root cause for the same crash.

The definition of the root cause is based on the program fix, which is knowledge that is unlikely to be available before the root cause is fixed—it is akin to having access to a perfect implementation. We now discuss how to achieve debug determinism without access to this perfect implementation.

Root-Cause Driven Selectivity

The definition of debug determinism suggests a simple strategy for achieving it in a real replay system: record or precompute just the root cause events and then use inference to fill in the missing pieces. However, the key difficulty with this approach is in identifying the root cause. One approach is to conservatively record or precompute all non-determinism (hence providing perfect determinism during replay), but this strategy results in high runtime overhead. Another approach is to leverage developer-provided hints as to where potential root causes may lie, but this is likely to be imprecise since it assumes *a priori* knowledge of all possible root causes.

To identify the root cause, we observe that, based on various program properties, one

can often *guess* with high accuracy where the root cause is located. This motivates our approach of using heuristics to detect when a change in determinism is required without actually knowing where the root cause is. We call this heuristic-driven approach *root cause-driven selectivity (RCSE)*. The idea behind RCSE is that, if strong determinism guarantees are provided for the portion of the execution surrounding the root cause and the failure, then the resulting replay execution is likely to be debug-deterministic. Of course, RCSE is not perfect, but further experimentation may reveal that it provides a close approximation of debug determinism. The following paragraphs present several potential variants of RCSE.

Code-Based Selection. This heuristic is based on the assumption that, for some application types, the root cause is more likely to be contained in certain parts of the code. For example, in datacenter applications like Bigtable, Chapter 3 argues that the control-plane code—the application component responsible for managing data flow through the system—is responsible for most program failures.

This observation suggests an approach in which we identify control-plane code and reproduce its behavior precisely, while taking a more relaxed approach toward reproducing data-plane code. Since control-plane code executes less frequently and operates at substantially lower data rates than data-plane code, this heuristic can reduce the recording overhead of a replay-debugging system. The key challenge is in identifying control-plane code, as the answer is dependent on program semantics. One promising approach was suggested in Chapter 3 of this dissertation: deem low-data rate code as control-plane, since data-plane code often operates at high data rates. The empirical results in that same chapter show that such automated control-plane selection has high accuracy for several typical datacenter applications, such as Hypertable and CloudStore.

Data-Based Selection. Data-based selection can be used when a certain condition holds on program state. For instance, if the goal is to reproduce a bug that occurs when a server processes large requests, developers could make the selection based on when the request sizes are larger than a threshold. Thus, high determinism will be provided for debugging failures that occur when processing large requests.

A more general approach is to watch for a set of invariants on program state: the moment the execution violates these invariants, it is likely on an error path. This is a signal to the RCSE system to increase the determinism guarantees for that particular segment of the execution. Ideally, assuming perfect invariants (or specification), the root cause and the events up to the failure will be recorded with the highest level of determinism guarantees. If such invariants are not available, we could use dynamic invariant inference [20] before the software is released. While the software is running in production, the replay-debugging system monitors the invariants. If the invariants do not hold, the system switches to high determinism recording, to ensure the root cause is recorded with high accuracy.

Combined Code/Data Selection. Another approach is to make the selection at runtime using dynamic triggers on both code and data. A trigger is a predicate on both code and data that is evaluated at runtime in order to specify when to increase recording granularity. An example trigger is a "potential-bug detector". Given a class of bugs, one can in many cases identify deviant execution behaviors that result in potential failures [51]. For instance, data corruption failures in multi-threaded code are often the result of data races. Low-overhead data race detection [30] could be used to dial up recording fidelity when a race is detected.

Therefore, triggers can be used to detect deviant behavior at runtime and to increase the determinism guarantees onward from the point of detection. The primary challenge with this approach is in characterizing and capturing deviant behavior for a wide class of root causes. For example, in addition to data races, data corruption may also arise due to forgetting to check system call arguments for errors, and increasing determinism for all such potential causes may increase overhead substantially. A compelling approach to create triggers is to use static analysis to identify potential root causes at compile time and synthesize triggers for them.

All heuristics described above determine when to dial up recording fidelity. However, if these heuristics misfire, dialing down recording fidelity is also important for achieving low-overhead recording. For code-based selection, we can dial down recording fidelity for data-plane code. For trigger-base selection, we can dial down recording fidelity if no failure is detected and no trigger fired for a certain period of time.

8.1.2 Assessing Determinism Model Utility

So far, work on replay-debugging has not employed metrics that evaluate debugging power. Instead, the comparison was mainly based on recording performance figures and ad-hoc evidence of usefulness in debugging. Instead, we propose a metric aimed at encouraging systematic progress toward improving debugging utility.

Debugging fidelity (*DF*) is the ability of a system to reproduce accurately the root cause and the failure. If a system does not reproduce the failure, debugging fidelity is 0, because developers cannot inspect how the system reaches failure. If the system reproduces the original root cause and the failure, debugging fidelity is 1. If the system reproduces the failure, but a different root cause from the original, debugging fidelity is $1/n$, where n is the number of possible root causes for the failure observed in the original execution. This definition takes into account the fact that a replayed execution is still useful for debugging even if it reproduces the failure through a different root cause, yet the replay is useless for debugging if it does not reproduce the failure.

It may be difficult to analytically determine a replay system's debugging fidelity. However, it is possible to determine it empirically. For instance, static analysis could be used to identify the location of all possible root causes for a certain failure, potentially including false positives. One can then manually weed out the false positives and check if the system can replay all of the true positives. Another approach is to empirically test if a replay-debugging

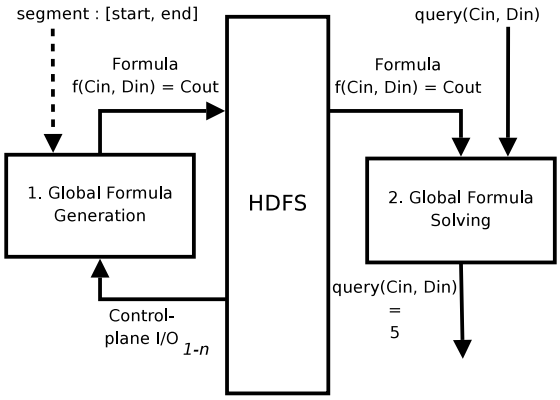

Figure 8.1: A closer look at DCR's Distributed-Replay Engine (DRE) with Just-In-Time Inference (JIT-DRI). JIT-DRI requires both an execution segment identifier (a starting and ending instruction number) and a query on execution state (e.g., variable x at instruction 2000).

system correctly replays in the cases when given root causes are guaranteed to be present in the original execution through some other means (e.g., deterministic execution).

Debugging efficiency (*DE*) is the duration of the original execution divided by the time the tool takes to reproduce the failure, including any analysis time. Normally this metric has values less than 1, but it is possible for techniques such as execution synthesis [52] to synthesize a substantially shorter execution. If this shorter execution compensates for post-factum analysis time, debugging efficiency can have values greater than 1.

Debugging utility (*DU*) is the product of debugging fidelity and debugging efficiency: $DU = DF \times DE$.

8.2 Practical Inference

DCR enables the developer to speed up inference time by recording more of the original run. However, the resulting in-production overheads may be too high to tolerate. Of course, the ideal configuration is just to record just control-plane I/O, and then have DRI infer all the unrecorded non-determinism. However, obtaining this ideal is hard because inference must explore the exponential space of all inputs and thread schedules. In this section, we describe ways of speeding inference time for each component of the inference phase under the ideal recording setting (i.e., recording just control plane I/O).

8.2.1 Just-in-Time Inference

Just-in-Time DRI (JIT-DRI) is a potential way to reduce long inference times. The key observation behind JIT-DRI is that developers are often interested in reasoning about only a small portion of the replay run—a stack trace here or a variable inspection there. For such usage patterns, it makes little sense to infer the concrete values of all execution states. For debugging then, it suffices to infer, in an on-demand manner, the values for just those portions of state that interest the user of the replay system.

Figure 8.1 (dashed and solid) illustrates the DRI architecture with the JIT optimization enabled. JIT DRI accepts an execution segment of interest and state expression from the debugger. The segment specifies a time range of the original run and can be derived by manually inspecting console logs. JIT DRI then outputs a concrete value corresponding to the specified state for the given execution segment.

JIT DRI works in two phases that are similar to non-JIT DRI. But unlike non-JIT DRI, each stage uses the information in the debugger query to make more targeted inferences:

JIT Global Formula Generation. In this phase, JIT-DRI generates a formula that corresponds only to the execution segment indicated by the debugger query.

The unique challenge faced by JIT FormGen is in starting the symbolic execution at the segment start point rather than at the start of program execution. To elaborate, the symbolic state at the segment start point is unknown because DRI did not symbolically execute the program before that. The JIT Formula Generator addresses this challenge by initializing all state (memory and registers) with fresh symbolic variables before starting symbolic execution, thus employing Engler's under-constrained execution technique [19].

For debugging purposes, under-constrained execution has its tradeoffs. First, the inferred execution segments may not be possible in a real execution of the program. Second, even if the segments are realistic, the inferred concrete state may be causally inconsistent with events (control-plane or otherwise) before the specified starting point. This could be especially problematic if the root-cause being chased originated before the specified starting point. We have found that, in practice, these relaxation are of little consequence so long as `DCR` reproduces the original control plane behavior.

JIT Global Formula Solving. In this phase, JIT-DRI solves only the portion of the previously generated formula that corresponds to the variables (i.e., memory locations) specified in the query.

The main challenge here is to identify the constraints that must be solved to obtain a concrete value for the memory location. We do this in two steps. First we resolve the memory location to a symbolic variable, and then we resolve the symbolic variable to a set of constraints in the formula. We perform the first resolution by looking up the symbolic state at the query point (this state was recorded in the formula generation phase). Then for the second resolution, we employ a connected components algorithm to find all constraints

related to the symbolic variable. Connected components takes time linear in the size of the formula.

Bibliography

[1] Cloudstore. `http://kosmosfs.sourceforge.net/`.

[2] Hypertable. `http://www.hypertable.org/`.

[3] Hypertable issue 63. http://code.google.com/p/hypertable/issues/.

[4] Memcached. `http://www.memcached.org/`.

[5] Openssh. `http://www.openssh.com/`.

[6] Vmware vsphere 4 fault tolerance: Architecture and performance, 2009.

[7] Altekar, G., and Stoica, I. Odr: output-deterministic replay for multicore debugging. In *SOSP* (2009).

[8] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage.

[9] Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS* (2010).

[10] Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., and Chau, J. Framework for instruction-level tracing and analysis of program executions. In *VEE* (2006).

[11] Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst. 3* (February 1985), 63–75.

[12] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).

[13] Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., and Bosschere, K. D. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet* (2003).

[14] Dean, J., and Ghemawat, S. Mapreduce: a flexible data processing tool. *CACM 53*, 1 (2010).

[15] Devietti, J., Lucia, B., Ceze, L., and Oskin, M. Dmp: deterministic shared memory multiprocessing. In *ASPLOS* (2009).

[16] Dionne, C., Feeley, M., and Desbiens, J. A taxonomy of distributed debuggers based on execution replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (Sunnyvale, CA, Aug. 1996).

[17] Dunlap, G. W., Lucchetti, D. G., Fetterman, M. A., and Chen, P. M. Execution replay of multiprocessor virtual machines. In *VEE* (2008).

[18] Ellithorpe, J. D., Tan, Z., and Katz, R. H. Internet-in-a-box: emulating datacenter network architectures using fpgas. In *DAC* (2009).

[19] Engler, D., and Dunbar, D. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA* (2007).

[20] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27*, 2 (Feb. 2001), 1–25.

[21] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for java. In *PLDI* (2002).

[22] Floyd, R. W. Assigning meaning to programs. 19–32.

[23] Ganesh, V., and Dill, D. L. A decision procedure for bit-vectors and arrays. In *CAV* (2007).

[24] Geels, D., Altekar, G., Maniatis, P., Roscoe, T., and Stoica, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[25] Geels, D., Altekar, G., Shenker, S., and Stoica, I. Replay debugging for distributed applications. In *USENIX* (2006).

[26] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *SOSP* (2003).

[27] Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M. F., and Zhang, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008).

[28] Huselius, J. Debugging parallel systems: A state of the art report. Tech. Rep. MDH-MRTC-63/2002-1-SE, Maelardalen Real-Time Research Centre, Sept. 2002.

[29] Intel. *Intel 64 and IA-32 Architectures Reference Manual*, November 2008.

[30] John Erickson, Madanlal Musuvathi, S. B., and Olynyk, K. Effective data-race detection for the kernel.

[31] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976), 385–394.

[32] Konuru, R. Deterministic replay of distributed java applications. In *IPDPS* (2000).

[33] Lahiri, S. K., Qadeer, S., and Rakamaric, Z. Static and precise detection of concurrency errors in systems code using smt solvers. In *CAV* (2009).

[34] LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Computers 36*, 4 (1987), 471–482.

[35] Lee, D., Said, M., Narayanasamy, S., Yang, Z., and Pereira, C. Offline symbolic analysis for multi-processor execution replay. In *MICRO* (2009).

[36] Lee, D., Wester, B., Veeraraghavan, K., Narayanasamy, S., Chen, P. M., and Flinn, J. Online multiprocessor replay via speculation and external determinism. In *ASPLOS* (2010).

[37] Liu, X., Lin, W., Pan, A., and Zhang, Z. Wids checker: Combating bugs in distributed systems. In *NSDI* (2007).

[38] Montesinos, P., Ceze, L., and Torrellas, J. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA* (2008).

[39] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS* (2009).

[40] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and reproducing heisenbugs in concurrent programs. In *OSDI* (2008).

[41] Newsome, J., and Song, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS* (2005).

[42] Olszewski, M., Ansel, J., and Amarasinghe, S. P. Kendo: efficient deterministic multithreading in software. In *ASPLOS* (2009), pp. 97–108.

[43] Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K. H., and Lu, S. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP* (2009).

[44] Ronsse, M., and Bosschere, K. D. Recplay: a fully integrated practical record/replay system. *ACM TOCS 17*, 2 (1999).

[45] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Oakland* (2010).

[46] Stone, J. M. Debugging concurrent processes: a case study. In *PLDI* (1988).

[47] Vogels, W. Keynote address. CCA, 2008.

[48] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (New York, February 1995), ACM Press, pp. 24–37.

[49] Xu, W., Huang, L., Fox, A., Patterson, D. A., and Jordan, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (2009).

[50] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

[51] Zamfir, C., and C, G. Low-overhead bug fingerprinting for fast debugging. In *Runtime Verification* (2010).

[52] Zamfir, C., and Candea, G. Execution synthesis: A technique for automated software debugging. In *EuroSys* (2010).