DCR : Replay-Debugging for the Datacenter

Gautam Altekar
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

We've built a tool for debugging non-deterministic failures in production datacenter applications. Our system, called DCR, is the first to efficiently record and replay large scale, distributed, and data-intensive systems such as HDFS/GFS, HBase/Bigtable, and Hadoop/MapReduce. The enabling idea behind DCR is that debugging doesn't require a precise replica of the original datacenter run. Instead, it suffices to produce some run that exhibits the original *control-plane behavior*. This report details the design and implementation of DCR and provides preliminary results.

1 Introduction

Recent years have seen the rise of *datacenter systems*–large scale, distributed, and data-intensive applications such as HDFS/GFS [17], HBase/Bigtable [6], and Hadoop/MapReduce [7]. These mammoth systems employ thousands of nodes, spread across multiple datacenters, to process terabytes of data per day. Companies like Facebook, Google, and Yahoo! already use these systems to process their massive data-sets. But an ever-growing user population and the ensuing need for new and more scalable services means that novel datacenter systems will continue to be built.

Unfortunately, debugging is hard, and we believe that this difficulty has impeded the development of existing and new datacenter systems. 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 [32]. Thus *effective tools for debugging non-deterministic failures in production datacenter systems are sorely needed.*

Developers presently use a range of methods for debugging non-deterministic failures. But they all fall short in the datacenter environment. The widely-used approach of manual 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 [10, 20, 28] can find the errors underlying several non-deterministic 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 [33] and diagnosing failures [34], but the inferences they draw are fundamentally limited by the fidelity of developer-instrumented console logs.

In this paper, we show that *replay-debugging technology* (a.k.a, deterministic replay, record/replay) can be used to effectively debug non-deterministic 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. These replay runs may then be analyzed using a conventional debugger (e.g., GDB) or more sophisticated automated analyses (e.g., race and memory-leak detection , global predicates [15, 25], causality tracing [14], etc.).

Many replay systems have been built over the years and experience indicates that they are invaluable in reasoning about non-deterministic failures [1, 4, 9, 15, 16, 24, 25, 30]. However, no existing system meets the unique demands of the datacenter environment. Classical replay systems such as liblog [16] and VMWare [1] require all non-deterministic data to be logged, including data-race outcomes and program inputs, hence incurring high throughput losses and storage costs on multicore, terabyte-quantity processing. Recent relaxed-deterministic replay systems such as PRES [30] and ReSpec [24] enable efficient multicore recording, but, like classical replay systems, still record all program inputs. Finally, inference-based systems such as ODR [2] (our prior work), ESD [35], and SherLog [34] support efficient multicore recording without recording program inputs, but may take exponential time to generate a replay execution.

To address these shortcomings, we have built `DCR`–a Data Center Replay system that records and replays production runs of datacenter systems like Cassandra, HDFS, Hadoop, HBase, and Hypertable. 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. The control plane tends to be complicated and thus serves as the breeding ground for many bugs in datacenter software. But at the same time, the control-plane often operates at very low data-rates. Hence, by reproducing just the control-plane behavior, `DCR` circumvents the need to record data-plane inputs, and consequently achieves low record overheads with tolerable sacrifices of replay fidelity.

The central challenge in building `DCR` is that of reproducing the control-plane behavior of a datacenter app without knowledge of its original data-plane inputs. This is challenging because the control-plane's behavior depends on the data-plane's behavior: a 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, we employ Deterministic-Run Inference (DRI) [2]–an offline inference mechanism we developed in previous work–to compute data-plane inputs consistent with the recorded control-plane input/output behavior of the original run. Once inferred, `DCR` then substitutes the data-plane inputs along with the recorded control-plane inputs into subsequent program runs to generate a control-plane deterministic run.

`DCR` is not the first replay-debugger to generate a relaxed deterministic replay run using an offline compute phase; Stone introduced the idea in 1988 [31] and recent relaxed-deterministic replay systems such as ODR [2] (our prior work), PRES [30], ReSpec [24], and ESD [35] largely build on her ideas. But these systems are either incapable of providing control-plane determinism (PRES and ReSpec), hence incurring high datacenter logging overheads, or have offline compute times that are exponential in the length of the original run (ODR and ESD), hence precluding their use on long-running datacenter apps. `DCR` also builds on Stone's ideas of relaxing determinism. But in contrast with other derivatives, `DCR` is the first to enable responsive debugging of control-plane deterministic runs shortly after the failure was observed–a property that is essential for practical replay-debugging of datacenter apps.

`DCR` achieves responsive debugging times through the use of Just-In-Time DRI (JIT DRI)–an optimized version of DRI that avoids reasoning about an entire run (an expensive proposition) before replay can begin. The key observation underlying 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. JIT DRI enables `DCR` to replay-debug datacenter systems in a timely manner.

1.1 Requirements

A practical replay-debugging system for datacenter applications must meet a challenging set of requirements.

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.

Always-on operation, as opposed to periodic use, is key because failures tend to be unpredictable and often manifest long after an underlying error has been triggered. Without knowledge of when these errors occur, we must work under the assumption that they may occur at any time. Secondly, using the system periodically risks perturbing the execution, and consequently, masking errors in that execution. But recording all the time would mean that any perturbations would become part of the normal system behavior.

In the datacenter, supporting always-on operation is difficult. The system should have minimal impact on production throughput (less than 10%). But most 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 DFS).

Whole-system replay. The system should be able to replay-debug *all nodes* in the distributed system, if desired, *shortly* after a failure was observed.

Whole-system replay is essential because a failure or its underlying error may occur on any node in the system, and unfortunately, we have no way of reliably anticipating the precise node(s) on which the error(s) will manifest. Of course, whole-system replay is useless if the developer can't begin replay-debugging nodes in a reasonable amount of time after the failure. This rules out exponential-time offline processing phases that must be completed before replay-debugging can begin (e.g., Stone [31], ODR [2], PRES [30], etc.).

Providing responsive whole-system 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 those nodes, replay-debugging cannot be provided.

Out-of-the-box use. The system should record and replay *arbitrary* user-level applications on modern commodity hardware *with no administrator or developer effort*. This means that it should not require special hardware, languages, or source-code analysis and modifications.

The commodity hardware requirement is essential because we want to replay existing datacenter systems as well as future systems[1]. Special languages and source-code modifications (e.g., custom APIs and annotations, as used in R2 [18]) are undesirable because they are cumbersome to learn, maintain, and retrofit onto existing datacenter apps. Source-code analysis (e.g., as done in ESD [35] and SherLog [34]) is also prohibitive due to the extensive use of dynamically generated (i.e., JITed) code and dynamically linked libraries. For instance, the Hotspot JVM, used by HDFS/Hadoop/HBase/Cassandra, employs dynamic compilation.

Language support and source-code modification/analysis provide rich semantic information about the application. Without this information, the task of determining what information to record, or in some cases what information is missing, becomes much harder. Certainly, we could record all sources of non-determinism. But doing so would conflict with our goal of providing always-on operation and whole-system replay.

1.2 Contributions

In short, we make four key contributions to the state of the art in replay-debugging. First, we have built `DCR`–the first system that meets all of the aforementioned requirements for replay-debugging production runs of datacenter apps. Second, we introduce the notion of control-plane determinism–the enabling concept underlying `DCR`. Third, we present Just-In-Time Inference–an essential optimization that makes offline inference of control-plane deterministic runs feasible. And finally, we provide initial results with `DCR` on real datacenter systems.

2 Overview

We've built a replay-debugging system, called `DCR`, that meets all the aforementioned requirements for replay-debugging production datacenter systems. We begin this section with the key insight behind `DCR`. Then we describe how we turn this insight into an approach and architecture.

2.1 Observation: The Control Plane is Key

The central observation behind `DCR` is that, for debugging datacenter apps, we do *not* need a precise replica of the original run. Rather, it generally suffices to reproduce *some* run that exhibits the original *control-plane* behavior .

The control-plane of a datacenter app is the code that manages or controls data-flow, like figuring out where a block is stored in a distributed filesystem, maintaining replica consistency in a meta-data server, or updating routing table entries in a software router. Control-plane operations tend to be complicated–they account for 90% of the newly-written code in datacenter software and serve, not surprisingly, as breeding-grounds for distributed race-condition bugs . On the other hand, the control-plane is responsible for only 1% of all datacenter traffic .

A corollary observation is that datacenter debugging rarely requires reproducing the same *data-plane* behavior. The data-plane of a datacenter app is the code that processes the data. Examples include code that computes the checksum of an HDFS filesystem block or code that searches for a string as part of a MapReduce job. In contrast with the control-plane, data-plane operations tend to be simple–they account for just 10% of the newly-written code in a datacenter app and are often part of well-tested libraries. Yet, the data-plane is responsible for generating 99% of datacenter traffic.

2.2 Approach: Control-Plane Determinism

2.2.1 Overview

The complex yet low data-rate nature of the control-plane motivates `DCR`'s *approach of relaxing its determinism guarantees*. Specifically, `DCR` aims for *control-plane determinism*–a guarantee that replay runs will exhibit identical control-plane behavior to that of the original

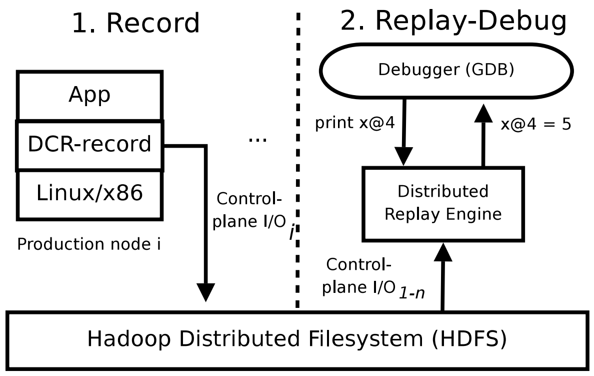

Figure 1: DCR's distributed architecture and operation.

run. Control-plane determinism enables datacenter replay because it circumvents the need to record data-plane communications (which have high data-rates), thereby allowing us to efficiently record and replay all nodes in the system without much programmer effort.

Figure 1 shows the architecture of our control-plane deterministic replay-debugging system. Like most replay systems, it operates in two phases.

Record mode. DCR records *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 communications performed by control-plane code. These communications may be between CPUs on different nodes (e.g., via sockets) or between CPUs on the same node (e.g., via shared memory). DCR streams the control-plane I/O to a Hadoop Filesystem (HDFS) cluster–a highly available distributed data-store designed for datacenter operation–using Chukwa [5].

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 control-plane I/O to provide the operator with a causally-consistent, control-plane deterministic view of the original distributed execution. The operator interfaces with the DRE using a distributed variant of GDB that we developed in prior work (see the Friday system [15]). Like GDB, our debugger supports inspection of local state (e.g., variables, backtraces). But unlike GDB, it also supports inspection of global state via global predicates .

3 Challenges

3.1 Recording Control Plane I/O

3.1.1 Overview

To record control-plane I/O, DCR must first identify it. Unfortunately, such identification generally requires a deep understanding of program semantics, and in particular, whether or not the I/O emanates from control-plane code.

In some cases, we can expect the developer to manually identify control-plane modules with little effort. For example, a developer will likely know that in HDFS the master node is a control-plane only module, and that almost all its I/O is control-plane in nature. Not all datacenter apps are so cleanly compartmentalized, however. Cassandra (Facebook's P2P storage system), for instance, intermixes control and data-plane code. And even those apps that are, at the high-level, purely data-plane (e.g., HDFS slaves) still have some control-plane code; for example to parse lookup requests from clients.

Rather than rely on the developer to understand the nuances of complicated systems software, DCR aims for automatic identification of control-plane I/O. The observation behind our identification method is that control and data plane I/O generally flow on distinct communication channels, and that each type of channel has a distinct signature. We leverage this observation to interpose on communication channels and then record the transactions (i.e., reads and writes) of only those channels that are classified as control-plane channels.

Of course, any classification of program semantics based on observed behavior will likely be imperfect. Nevertheless, our experimental results show that, in practice, our techniques provide a tight over-approximation–enough to eliminate developer burden and be considered useful.

3.1.2 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. The channels we consider not only include explicitly defined channels such as sockets, pipes, tty, and file I/O, but also implicitly defined channels such as message header channels (e.g., the first 32 bytes of every message) and shared memory.

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, least because it requires intercepting every load and store. One could improve performance by considering accesses to only shared pages. But this too incurs high overheads in multi-threaded apps (i.e., most datacenter apps) where the entire 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 [22] and later implemented and refined by SMP-ReVirt [9]. 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.

Page-based CREW in the context of deterministic replay has been well-documented by the SMP-ReVirt system [9], so we omit details here. However, we note that `DCR`'s use of CREW differs from that of SMP-ReVirt's in two major ways. First, `DCR` isn't interested in recording the ordering of accesses, but rather the content of each page (assuming the page is in the control-plane); on a CREW fault, `DCR` records the faulting page (4K on x86-32), just like it would had it received a page's worth of data on a control-plane socket. Second, `DCR` is interested only in user-level sharing (it's a user-level replay system), so false-sharing in the kernel (e.g., due to spin-locks) isn't an issue for us (false-sharing at user space is, though; see Section 3.1.4).

3.1.3 Classifying Channels

We use the channel's data-rate to identify its type. That is, if the channel data-rate exceeds a threshold, then we deem it a data-plane channel and stop recording it. If not, then we treat it as a control-plane channel and record it. The data-rate metric is effective because control-plane code generally operates at low data-rates. This makes sense because the goal of the control-plane is to coordinate and manage workloads, not to process or transfer it.

Socket, pipe, tty, and file channels. The data-rates on these channels are measured in bytes per second. `DCR` measures these rates by keeping track of the number of bytes transferred (as indicated by `sys_read()` return values) over time. We maintain a simple moving average over a t-second window, where $t = 2$ by default.

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 collect the page-fault rate by updating a counter on each CREW fault, and maintaining a moving average of a 1 second window. We chose a control-plane threshold of 25 faults/sec.

Though effective, the strategy of using CREW page-faults to detect shared-memory communication is not without drawbacks. 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. In practice, the impact of this drawback is mitigated by the fact that user-level apps (especially those that use `pthreads`) rarely employ spin-locks. In particular, `pthread_mutex_lock()` will await notification of lock availability in the kernel rather than spin incessantly.

3.1.4 Avoiding High CREW Fault-Rates

The CREW protocol, under certain workloads, can incur high page-fault rates than in turn will seriously degrade performance (see SMP-ReVirt). Often this is due to legitimate sharing between CPUs, such as when CPUs contend for a spin-lock. Sometimes, however, the sharing may be false–a consequence of unrelated data-structures being housed on the same page. In such circumstances, CPUs aren't actually communicating on a channel.

Regardless of the cause, `DCR` employs a simple strategy to avoid high page-fault rates. When `DCR` observes that the fault-rate threshold for a page is exceeded (i.e., is a data-plane channel), it removes all page protections from that page and subsequently enables unbridled access to it, thereby effectively turning CREW off for that page. CREW is then re-enabled for the page n seconds in the future to determine if data-rates have changed.

3.2 Providing Control-Plane Determinism

3.2.1 Distributed Deterministic-Run Inference

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 debugger queries. This is challenging because, although `DCR` knows the original control-plane inputs, it does not know

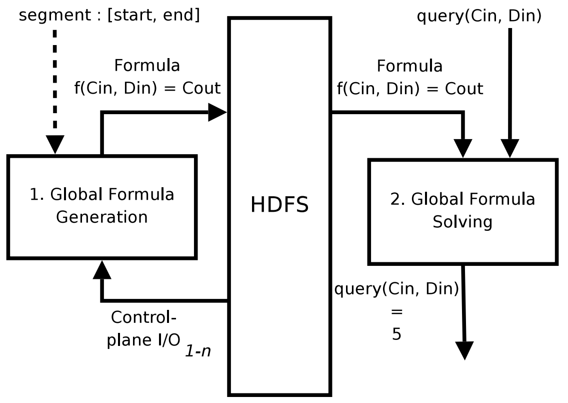

Figure 2: A closer look at DCR's Distributed-Replay Engine (DRE). It employs Distributed Deterministic-Run Inference to provide the debugger with a control-plane deterministic view of distributed state. With the Just-In-Time optimization enabled, the DRE requires an additional query argument (dashed).

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 Distributed Deterministic Run Inference (DDRI)–the distributed extension of a single-node inference mechanism we previously developed to efficiently record multiprocessor execution (see the ODR system [2]). DDRI leverages the original run's control-plane I/O (previously recorded by DCR) and program analysis to compute a control-plane deterministic view of the query-specified program state. DDRI's program analysis operates entirely at the machine-instruction level and does not require annotations or source-code. Moreover, it works on large and sophisticated datacenter software–even those that run on the JVM (notorious for dynamically generating code).

Depicted in Figure 2, DDRI works in two stages. In the first stage, *global formula generation*, DDRI translates the distributed program into a logical formula that represents the set of all possible distributed, control-plane deterministic runs. Of course, the debugger-query isn't interested in this set. Rather, it is interested in a subset of a node's program state from just one of these runs. So in the second phase, *global formula solving*, DDRI dispatches the formula to a constraint solver. The solver computes a satisfiable assignment of variables for the unknowns in the formula, thereby instantiating a control-plane deterministic run. From this run, it then extracts and returns the debugger-requested execution state.

3.2.2 Global Formula Generation

Generating a single formula that captures the behavior of a large scale datacenter system is hard, for two key reasons. First, a datacenter system is composed of hundreds of CPUs, and the formula must capture all of their behaviors. Second, the behavior of any given CPU in the system depends on the behavior of other CPUs. Thus the formula needs to capture the collective behavior of the system so that inferences we make from the formula are causally consistent across CPUs.

To capture the behavior of multiple, distributed CPUs, DCR generates a *local formula* for each CPU. A local formula for CPU i, denoted as $L(Cin_i, Din_i) = Cout_i$, represents the set of all control-plane deterministic runs for that CPU, independent of the behavior of all other CPUs. DCR knows the control-plane inputs (Cin_i) of all CPUs, so the only unknowns in the formula are the CPU's data-plane inputs (Din_i). Local formula generation is distributed on available nodes in the cluster and is described in further detail in Section 3.2.4.

To capture the collective behavior of distributed CPUs, DCR binds the per-CPU local formulas (L_i's) into a final global formula G. The binding is done by taking the logical conjunction of all local formulas and a *global causality condition*. The global causality condition is a set of constraints that requires any message received by a CPU to have a corresponding and preceding send event on another CPU, hence ensuring that inferences we make from the formula are causally consistent across nodes. In short, $G = L_0 \wedge \ldots \wedge L_n \wedge C$, where C is the global causality condition.

3.2.3 Global Formula Solving

In theory, DDRI could send the generated global formula, in its entirety, to a lone constraint solver. However, in practice, this strategy is doomed to fail as modern constraint solvers are incapable of solving the multi-terabyte formulas and NP-hard constraints produced by sophisticated and long-running datacenter apps. Section 3.3.1 discusses how we address this challenge.

3.2.4 Local Formula Generation

DDRI translates a program into a local-formula using Floyd-style verification condition generation [13]. The DDRI generator most resembles the generator employed by Proof-Carrying Code (PCC) [29] in that it works by symbolically executing the program at the instruction level, and produces a formula representing execution along multiple program paths. However, because the

PCC and DDRI generators solve different problems, they differ in the following ways:

Conditional and indirect jumps. Upon reaching a jump, the PCC generator will conceptually fork and continue symbolic execution along all possible successors in the control-flow graph. But when the jump is conditional or indirect, this strategy may yield formulas that are exponential in the size of the program.

By contrast, the DDRI generator considers only those successors implied by the recorded control-plane I/O. This means that when dealing with control-plane code, DDRI is able to narrow the number of considered successors down to one. Of course, the jump may be data-plane dependent (e.g., data-block checksumming code). In that case, multiple static paths must still be considered.

Loops. At some point, symbolic execution will encounter a jump that it has seen before. Here PCC stops symbolically executing along that path and instead relies on developer-provided loop-invariant annotations to summarize all possible loop iterations, hence avoiding "path explosion" .

Rather than rely on annotations, DDRI unrolls the loop a fixed number of times (similar to the unrolling done by ESC-Java [12]), using the branch count of the next recorded system event (e.g., syscall) to establish an upper-bound. This effectively offloads the work of finding the right dynamic path through the loop to the constraint solver, hence avoiding path explosion during the generation phase (the solving phase is still susceptible, but see Section 3.3.1).

Indirect accesses (e.g., pointers). Dereferences of symbolic pointers may access one of many locations. To model this precisely, PCC models memory as a symbolic array, hence offloading alias analysis to the constraint solver. Though such offloading can scale with PCC's use of annotations, DDRI's annotation free requirement results in an intolerable burden on the constraint solver .

Rather than model all of memory as an array, we model only those pages that may have been accessed in the original run by the symbolic dereference. We know what those pages are because we recorded their IDs in the original run using conventional page-protection techniques .

Dynamically generated code. To symbolic execute along the program's control-flow graph, PCC (and others) assumes the availability of all program code before symbolic execution can begin. But this assumption breaks when dynamically linked libraries or JITed code enter the picture, as such code is not generated until run-time.

DDRI handles dynamically generated code by generating the control-flow graph on-demand as executable code pages are mapped into the address space. Of course, this assumes that code pages will be mapped in at the same times and with the same content during replay, the latter of which may not hold for symbolic code pages arising from data-plane tainted dynamic generation code (unlikely, but possible). We circumvent this problem, however, by ensuring that memory operations are deterministically replayed and that all mapped code pages have the same code as executed in the original run. `DCR` records this information in the original run with minimal overhead.

3.3 Shortening Debugger Response Time

A primary goal of `DCR` is to provide responsive interactive replay-debugging. Specifically, `DCR` aims for response times that are at most linear in the length of the original run. But to achieve this goal, DDRI (the post-run inference method introduced in Section 3.2) must surmount major scalability challenges.

3.3.1 Huge Formulas, NP-Hard Constraints

When used on datacenter systems, DDRI produces formulas that are intractably difficult for modern solvers. DDRI-generated formulas may be terabytes in size since DDRI must reason about long-running data-processing code that handle terabytes of data that `DCR` never recorded. But more fundamentally, these formulas often contain NP-hard constraints. This is not surprising as datacenter apps often invoke cryptographic routines (e.g., SHA-1) to perform off critical-path verification.

To overcome this challenge, we've developed Just-In-Time DDRI (JIT-DDRI)–an on-demand variant of DDRI that enables responsive inference-based debugging of datacenter apps. The observation underlying JIT-DDRI is that, when debugging, developers observe only a portion of the execution–a variable inspection here or a stack trace there. Rarely do they inspect all program states. This observation then implies that there is no need to solve the entire formula generated by FormGen, as that corresponds to the entire execution. Instead, *it suffices to solve just those parts of the formula that correspond to developer interest*.

Figure 2 (dashed and solid) illustrates the DDRI architecture with the JIT optimization enabled. JIT DDRI 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 DDRI then outputs a concrete value corresponding to the specified state for the given execution segment.

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

JIT Global Formula Generation. In this phase, JIT-DDRI 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 DDRI 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 [11].

For debugging purposes, under-constrained execution has its drawbacks. In particular, 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.

JIT Global Formula Solving. In this phase, JIT-DDRI solves only the portion of the previously generated formula that corresponds to the 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 FormGen 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 length of the formula.

3.3.2 Channel Causality

A replay-debugger is of limited use if it doesn't let the developer backtrack the chain of causality from the failure to its root cause. But guaranteeing causality in inferred datacenter runs is hard: it requires efficiently reasoning about communications spanning thousands of CPUs possibly spread across thousands of nodes. JITI (described in Section) can help with such reasoning by solving only those constraints involved in the chain of causality of interest to the developer. But if the causal chain is long, then even JITI-produced constraints may be overwhelmingly large for the solver.

We once again invoke the central theme of this work to overcome this challenge. Specifically, we observe that control-plane channel causality is essential for datacenter debugging and that data-plane causality is dispensable. Our approach then is to record causality of control-plane channels, but to forego recording or inferring causally consistent data-plane communications. The result of this approach is that we completely avoid reasoning about channel causality, as the causality that we care about (control-plane causality) is reproduced exactly as recorded.

The primary challenge with our approach is in recording control-plane causality. `DCR` captures causality using logical clocks , like most other distributed replay systems (see liblog [16] and R2 [18]). In explicitly defined channel such as sockets, clock values are piggybacked on channel messages, and updated per standard Lamport clock rules [21]. Message fragmentation in TCP/IP channels presents some challenges, but we leverage our prior work to overcome them (see the liblog replay system [16]). For shared memory channels, the clock values are transmitted through shared memory itself on CREW-faults, as done by SMP-ReVirt [9].

4 Implementation

`DCR` currently runs on Linux/x86. It consists of 120 KLOC of code (95% C, 3% ASM). 70 KLOC is due to the LibVEX binary translator. We developed the other 50 KLOC over a period of 6 person-years. Here we present a selection of the implementation challenges we faced.

4.1 User-Level Architecture

We designed `DCR` to work entirely at user-level for several reasons. First, we wanted a tool that works with and without VMs. After all many important datacenter environments don't use VMs. Secondly, we wanted the implementation to be as simple as possible. VM-level operation would require that the DRE reason about kernel behavior as well–a hard thing to get right. Moreover it avoids semantic gap issues . Finally, we found that interposing on control-plane channels to be efficient. Specifically, we were able to Linux's vsyscall page to avoid traps. Moreover we avoided high CREW fault rate due to false-sharing in the kernel.

Implementing the CREW protocol at user-level presented some challenges, primarily because Linux doesn't permit per-thread page protections (i.e., all threads share a page-table). This means that we can't turn off protections for a thread executing on one CPU while enable its for a thread running on a different CPU. We address this problem by extending each process's page table (by modifying the kernel) with per-CPU page-protection flags. When a thread gets scheduled in to a CPU, then it uses the protections for the corresponding CPU.

4.2 Formula Generation

DDRI generates a formula by symbolically executing the target program (see Section 3.2.4), in manner very similar to that of the Catchconv symbolic execution tool [26]. Specifically, symbolic execution proceeds at the machine instruction level with the aid of the LibVEX binary translation library. VEX translates x86 into a RISC-style intermediate language once basic block at a time. DDRI then translates each statement in the basic block to an STP constraint.

DCR's symbolic executor borrows several tricks from prior systems. An important optimization is constraint elimination, in which constraints for those instructions not tainted by symbolic inputs (e.g., data-plane inputs) are skipped.

4.3 Debugger Interface

DCR's debugger enables the developer to inspect program state on any node in the system. It is implemented as a Python script that multiplexes per-node GDB sessions on to a single developer console, much like the console debugger of the Friday distributed replay system [15]. With the aid of GDB, our debugger currently support four primitives: backtracing, variable inspection, breakpoints, and execution resume. Watchpoints and state modification are currently unsupported .

Getting DCR's debugger to work was hard because GDB doesn't know how to interface with the DRE. That is, unlike classical replay mechanisms, the DRE doesn't actually replay the application; it merely infers specified program state. However, the key observation we make is that GDB inspects child state through the `sys_ptrace` system call. This leads to DCR's approach of intercepting GDB's ptrace calls and translating them into queries that the DRE can understand. When the DRE provides an answer (i.e., a concrete value) to DCR, it then returns that value to GDB through the ptrace call.

5 Evaluation

5.1 Performance

The goal of this section is to provide a comparative evaluation of DCR's performance. A fair comparison, however, is difficult because, to our knowledge, no other publicly available, *user-level* replay system is capable of deterministically replaying the datacenter apps in our suite. Rather than compare apples with oranges, we base our comparison on a modified version of DCR, called C&D, that records both control and data plane non-determinism in a fashion most similar to SMP-ReVirt [9]–the state of the art in classical multi-core deterministic replay.

In short, we found that DCR incurs very low recording overheads (at about 15% slowdown and 3 GB/day log rates) suitable for always-on production use. Moreover, we found that DCR's debugger response times, though sometimes sluggish, are generally fast enough to be useful–even in the presence of NP-hard constraints. By contrast, C&D provides extremely responsive debugging sessions as would be expected of a classical replay system. But it incurs impractically high record-mode overheads (over 50% slowdown and 3 TB/day log rates) on datacenter workloads.

5.1.1 Setup

We evaluate six widely-used datacenter applications: *Cassandra*, *HBase*, *Hyptertable*, *HDFS*, *CloudStore*, and *Hadoop*. *Cassandra*, *HBase*, and *Hypertable* are distributed databases that are in production use at companies like Facebook and Twitter. *HDFS* and *CloudStore* are distributed filesystems that are in production use at companies like Yahoo! and BaiDu. *Hadoop* is a distributed data processing system in the style of MapReduce used by Yahoo! and others.

All apps were run on a 50-VM cluster on Amazon's EC2 , with workloads chosen to mimic peak datacenter operation and to finish in 10 minutes. Specifically, for the distributed databases, 25 clients performed concurrent lookups and deletions to a 20 terabyte table of web data. For the distributed filesystems, 25 clients performed concurrent gets and puts of 100 gigabyte files. Finally, *Hadoop* was made to perform a distributed grep of a 20 terabyte text file.

Each VM in our cluster operates at 2.0GHz and has 4GB of RAM. The OS used was Debian 5 with a 2.6.29 Linux kernel. The kernel was patched to support DCR's interpositioning hooks. 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.

5.1.2 Record Mode

Logging Rates. Figure 3 gives results for the record rate, a key performance metric for datacenter workloads. At the high level, the graph shows that DCR's log rates are suitable for the datacenter: they are less

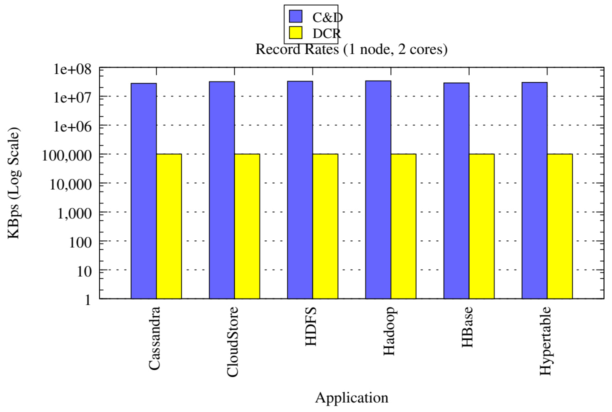

Figure 3: Record rates for C&D and DCR. DCR's rate is two orders-of-magnitude lower because it logs just the control-plane I/O.

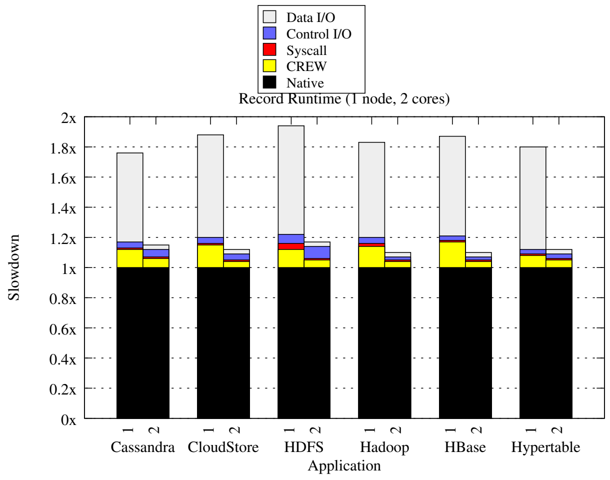

Figure 4: Record runtimes, normalized with native application execution time, for (1) C&D which records control and data planes and (2) DCR which records just the control plane. DCR's performance is as high as 60% better.

than those of traditional console logs and two orders of magnitude lower than C&D's rates (3 GB/day v. 3 TB/day). This result is not surprising because, unlike C&D, DCR does not record data-plane channels. Interestingly, shared-memory I/O plays a much more prominent role in DCR than in C&D. That's because DCR logs the contents of the entire page on each CREW fault while C&D logs only their ordering (in the style of SMP-ReVirt). The impact of recording 4K pages should be greater, but DCR deems pages with high fault-rates to be data-plane channels and stops recording them.

Slowdown. Figure 4 gives the slowdown incurred by DCR broken down by various instrumentation costs. At about 15%, DCR's record-mode slowdown is as much as 60% less than C&D's. Since DCR records just the control-plane, it doesn't have to compete for disk bandwidth with the application as DCR has to do. Overall, the DCR's slowdowns on data-intensive workloads are similar to those of classical replay systems on data-unintensive workloads.

DCR's slowdowns are greater than our goal of 1%. The main bottleneck is shared-memory channel interpositioning. Here CREW faults and the associated copying of the faulting page to the log are to blame. For some apps, pages may fault more than a thousand times a second (see SMP-ReVirt [9]). But fortunately, DCR avoids massive slowdowns by disabling CREW for the high data-rate pages.

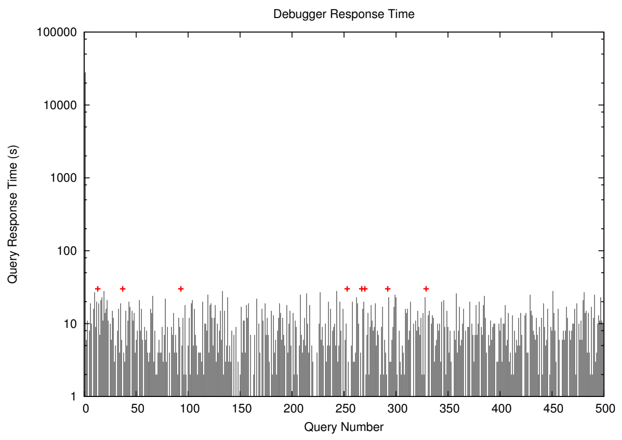

Figure 5: DCR's replay-debugging query response times for a 50 node HDFS cluster. The initial query is really slow, but subsequent queries are generally much faster. Dots denote queries that timed out.

5.1.3 Replay-Debug Mode

Figure 5 gives the debugger query-response times for DCR. C&D's response times are on the order of microseconds, as expected, and thus are omitted.

We collected DCR's response times using a script that simulates a manual debugging session. The script makes 500 queries for state from the first 10 minutes of the original distributed execution. The queries are directed at randomly selected nodes and may ask the replay engine to print a backtrace, return a variable's value (chosen from the stack context indicated by the backtrace), or step forward n instructions on those node. Non-initial queries that take longer than 10 seconds are cancelled.

Looking at Figure 5, DCR's response times have several notable features:

The first query is really slow, because it induces DDRI to generate a formula and then split it. Formula

	Always-on operation	Whole-system replay	Out-of-the-box use	Determinism
Hardware support [8, 23, 27], CoreDet [3]	*No*	*No*	*No*	Value[2]
liblog [16], VMWare [1], PRES [30], ReSpec [24]	*No*	*No*	Yes	I/O[3]
ESD [35], SherLog [34]	Yes	*No*	*No*	Output[4]
ODR [2]	Yes	*No*	Yes	Output
DejaVu [19]	Yes	*No*	Yes[5]	Value
R2 [18]	Yes	Yes	*No*	Value
DCR	**Yes**	**Yes**	**Yes**	**Control**

Figure 6: A comparison with other replay-debugging systems. Only DCR meets all the requirements for *datacenter apps*.

generation and split time are on average 49x and 9x, respectively, the runtime of the segment being debugged (10 minutes in this case). Though slow, both operations take time linear in the length of the segment.

Non-initial queries are generally fast, because results of the formula generation and splitting done in the first query were cached and reused in subsequent queries. The bulk of the effort for these queries is devoted to constraint solving. The solving time is generally fast because only the query-relevant portion of the original formula (as determined in the splitting phase) is solved. Some queries don't even require constraint solving because they are directed at control-plane state, which is concrete.

Some non-initial queries timeout, because the constraints they resolve to are fundamentally hard to solve, or because the constraint solver has a hard time with it (e.g., because it was non-linear). Note than in either case, the response times of subsequent queries remain unaffected–a key benefit of JIT formula solving.

6 Related Work

Figure 6 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).

Several relaxed-deterministic replay systems (e.g., Stone [31], PRES [30], and ReSpec [24]) and hardware and/or compiler assisted systems (e.g., Capo [27], Lee et al. [23], DMP [8], CoreDet [3], 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-system replay. Several replay systems can provide whole-system replay for small clusters, but not for large-scale, failure-prone datacenters. Specifically, systems such as liblog [16], Friday [15], VMWare [1], Capo [27], PRES [30], and ReSpec [24] 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 provide even partial-system replay 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 ODR [2] (our prior work), ESD [35], and SherLog [34] can efficiently replay some single-node apps (ESD more so than the others). But these systems were not designed for distributed operation, much less datacenter apps. Indeed, even single-node replay is a struggle for these systems. On long-running and sophisticated datacenter apps (e.g., JVM-based apps), they require reasoning about an exponential number of program paths, not to mention NP-hard computations, before a replay-debugging session can begin.

Out-of-the-box use. Several replay schemes employ hardware support for efficient multiprocessor recording . These schemes don't address the problem of efficient datacenter recording, however. 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 [3], ESD [35], and SherLog [34] employ C source code analyses to speed the inference process. However, applying such analyses in the presence of dynamic code generation and linking is still an open problem. Unfortunately, most of the datacenter apps we consider run within the JVM, well-known for dynamically generating code.

The R2 system [18] 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 datacenter recording overheads. Alas, such annotations are hardly "out of the box". They require considerable developer effort to manually identify the control-plane and to retrofit existing code bases.

7 Conclusion

We've presented DCR, a replay-debugging system for datacenter applications. We believe it is the first viable replay debugging system in that provides always-on operation, whole distributed system replay, and out of the box operation. The key idea behind DCR is control-plane determinism–the notion that it suffices to reproduce the control-plane behavior of the datacenter system. Coupled with Just-In-Time Inference, DCR enables practical replay-debugging of large-scale, data-intensive distributed systems. Looking forward, we hope to further improve DCR's recording overheads and debugging response times.

References

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

[2] ALTEKAR, G., AND STOICA, I. Odr: output-deterministic replay for multicore debugging. In *SOSP* (2009).

[3] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., AND GROSSMAN, D. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS* (2010).

[4] 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).

[5] BOULON, J., KONWINSKI, A., QI, R., RABKIN, A., YANG, E., AND YANG, M. Chukwa, a large-scale monitoring system. In *CCA* (2008).

[6] 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).

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: a flexible data processing tool. *CACM 53*, 1 (2010).

[8] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. Dmp: deterministic shared memory multiprocessing. In *ASPLOS* (2009).

[9] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *VEE* (2008).

[10] ELLITHORPE, J. D., TAN, Z., AND KATZ, R. H. Internet-in-a-box: emulating datacenter network architectures using fpgas. In *DAC* (2009).

[11] ENGLER, D., AND DUNBAR, D. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA* (2007).

[12] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for java. In *PLDI* (2002).

[13] FLOYD, R. W. Assigning meaning to programs. 19–32.

[14] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007).

[15] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[16] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *USENIX* (2006).

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003).

[18] 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).

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

[20] LAHIRI, S. K., QADEER, S., AND RAKAMARIC, Z. Static and precise detection of concurrency errors in systems code using smt solvers. In *CAV* (2009).

[21] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[22] LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. Debugging parallel programs with instant replay. *IEEE Trans. Computers 36*, 4 (1987), 471–482.

[23] LEE, D., SAID, M., NARAYANASAMY, S., YANG, Z., AND PEREIRA, C. Offline symbolic analysis for multi-processor execution replay. In *MICRO* (2009).

[24] 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).

[25] LIU, X., LIN, W., PAN, A., AND ZHANG, Z. Wids checker: Combating bugs in distributed systems. In *NSDI* (2007).

[26] MOLNAR, D. A., AND WAGNER, D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Tech. Rep. UCB/EECS-2007-23, EECS Department, University of California, Berkeley, 2007.

[27] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS* (2009).

[28] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *OSDI* (2008).

[29] NECULA, G. C., AND LEE, P. Safe kernel extensions without run-time checking. In *OSDI* (1996).

[30] 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).

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

[32] VOGELS, W. Keynote address. CCA, 2008.

[33] 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).

[34] 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).

[35] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated software debugging. In *EuroSys* (2010).

Notes

[1]Note that it's not clear at the moment what hardware would be needed to support datacenter replay–existing work on hardware support for efficient multi-core recording is of limited use in this domain.

[2]Lee at al. also aim for order determinism, though this is not guaranteed.

[3]PRES and ReSpec aim for I/O and partial-order determinism while liblog and VMWare provide value-determinism.

[4]SherLog provides best-effort console log output determinism–no guarantees are made.

[5]DejaVu was designed for Java apps, but its core techniques may be adapted in a more generic system.