DCR: Replay Debugging for the Datacenter

Gautam Altekar
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-74
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-74.html

May 13, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

DCR : Replay Debugging for the Datacenter

Gautam Altekar
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

Debugging is hard, but debugging production datacenter applications such as Cassandra, Hadoop, and Hypertable is downright daunting. 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 replay debugging 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 behaviors 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 all key requirements of production datacenter applications: lightweight recording of long-running programs, causally consistent replay of large scale systems, and out of the box operation on real-world applications.

1 Introduction

The past decade has seen the rise of large scale, distributed, data-intensive applications such as HDFS/GFS [26], HBase/Bigtable [13], and Hadoop/MapReduce [15]. These applications run on thousands of nodes, spread across multiple datacenters, and 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 applications will continue to be built.

Unfortunately, debugging is hard, and we believe that this difficulty has impeded the development of existing and new large scale distributed applications. 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 [42]. 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 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 [19, 30, 37] 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 [43] and diagnosing failures [44], 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) 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 conventional tracing tools (e.g., GDB and DTrace [12]) or more sophisticated automated analyses (e.g., race and memory-leak detection , global predicates [24, 34], and causality tracing [23]).

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 [5, 10, 18, 24, 25, 33, 34, 36, 39, 45]. However, no existing system meets the 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 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 distributed file system).

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

Providing 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. 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 (e.g., as done in ESD [45] and SherLog [44]) 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, and Cassandra, employs dynamic compilation.

1.2 Contributions

To meet all of the aforementioned requirements, we've built `DCR`–a Data Center Replay system that records and replays production runs of datacenter systems like Cloudstore, HDFS, Hadoop, HBase, and Hypertable . `DCR` contributes and leverages three key techniques.

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 often consists of millions of lines of source code [14]–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 [6]. 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.

Distributed 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: 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, `DCR` employs Distributed Deterministic-Run Inference (DDRI)–the distributed extension of an offline inference mechanism we developed in previous work [7]–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.

Just-in-Time Debugging. Though DCR is the first debugger to generate a relaxed deterministic replay session for datacenter applications, it is not the first to leverage the concept of an offline compute phase [7, 33, 39, 40, 45]. Unfortunately, this compute phase may take exponential time to finish in these predecessor systems. A large path space and NP-hard constraints are usually to blame. Regardless, a debugging session cannot be started until this phase is complete. By contrast, DCR can start a debugging session in time polynomial with the length of the original run.

DCR achieves a low time-till-debug through the use of Just-In-Time DDRI (JIT-DDRI)–an optimized version of DDRI that avoids reasoning about an entire run (an expensive proposition) before replay can begin. The key observation underlying JIT-DDRI 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.

2 Overview

We begin this section with the key insight behind DCR. Then we describe how we turn this insight into an approach.

2.1 Observation: The Control Plane is Key

The central observation behind DCR is that, for debugging datacenter applications, 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 application is the code that manages or controls data-flow. Examples of control-plane operations are locating a particular block 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 over 90% of the newly-written code in datacenter software [14] and serve, not surprisingly, as breeding-grounds for distributed race-condition bugs. On the other hand, the control-plane is responsible for only 5% of all datacenter traffic [6].

A corollary observation is that datacenter debugging rarely requires reproducing the same *data-plane* behavior. The data-plane of a datacenter application 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 under 10% of the code in a datacenter application [14] and are often part of well-tested libraries. Yet, the data-plane is responsible for generating and processing 95% of datacenter traffic [6].

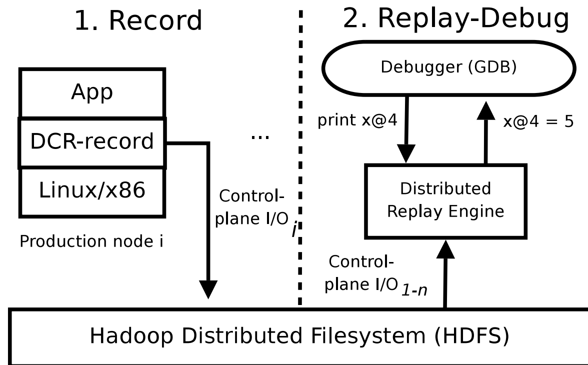

Figure 1: DCR's distributed architecture and operation. It uses the recorded control-plane I/O to answer debugger queries.

2.2 Approach: Control-Plane Determinism

The complex yet low data-rate 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 run. Control-plane determinism enables datacenter replay because it circumvents the need to record data-plane communications (which have high data-rates), thereby allowing DCR to efficiently record and replay all nodes in the system.

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 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 streams the control-plane I/O to a Hadoop Filesystem (HDFS) cluster–a highly available

distributed data-store designed for datacenter operation–using Chukwa [11].

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 replay system [24]). Like GDB, our debugger supports inspection of local state (e.g., variables, backtraces). But unlike GDB, it provides support for distributed breakpoints and global predicates–facilities that enable global invariant checking.

3 Design

In this section we present the key challenges of efficiently recording and replaying datacenter applications, and describe how we overcame them.

3.1 Recording Control Plane I/O

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.

Rather than rely on the developer to annotate and hence understand the nuances of sophisticated systems software, DCR aims for automatic identification of control-plane I/O. The observation behind DCR's 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. DCR leverages 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.

3.1.1 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. 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, 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 [31] and later implemented and refined by SMP-ReVirt [18]. 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 [18], so we omit the details here. However, we note that DCR's use of CREW differs from that of SMP-ReVirt's in two major ways. First, rather than record the ordering of accesses, DCR records the content of each access (assuming the access is to the control-plane). 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 spinlocks) isn't an issue for us (false-sharing at user space is, though; see our TR [8] for details on how we handle this).

3.1.2 Classifying Channels

As a simple heuristic, DCR uses the channel's data-rate to identify its type. That is, if the channel data-rate exceeds a threshold, then DCR deems it a data-plane

channel and stops recording it. If not, then DCR treats it as a control-plane channel and records it. The control-plane threshold for a channel is chosen using a token bucket algorithm [41]. That is, it is dynamically computed such that aggregate thresholds of all channels do not exceed the per-node logging rate (100 KBps in our trials). This simple scheme is effective because control-plane channels, though bursty, generally operate at low data-rates.

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. DCR collects the page-fault rate by updating a counter on each CREW fault, and maintaining a moving average of a 1 second window. DCR caps the per-node control-plane threshold for shared memory channels at 10K faults/sec. A larger cap can incur slowdowns beyond 20% (see [17] for the impact of CREW fault-rate on run time).

Though effective in practice, the heuristic of using CREW page-fault rate to detect control-plane shared-memory communication can lead to false negatives. 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, 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.

3.2 Providing Control-Plane Determinism

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 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.

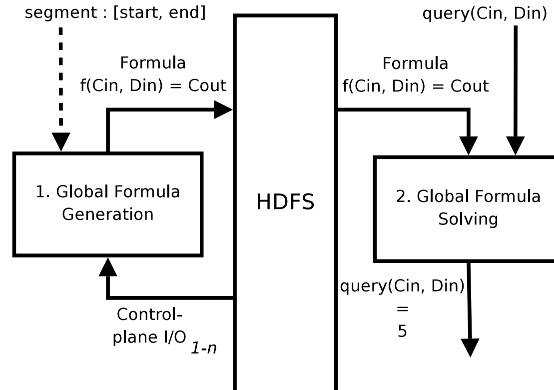

Figure 2: A closer look at DCR's Distributed-Replay Engine (DRE). It employs Distributed Deterministic-Run Inference to provide the debugger with a control-plane deterministic view of distributed state. With the Just-In-Time optimization enabled, the DRE requires an additional query argument (dashed).

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 replay system [7]). 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.

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, DDRI then extracts and returns the debugger-requested execution state.

3.2.1 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 may be composed of thousands of CPUs, and the formula must capture all of their behaviors. Second, the behavior of any given

CPU in the system may depend 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 $L_i(Cin_i, Din_i) = Cout_i$, 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 I/O (Cin_i and $Cout_i$) 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 3.2.3.

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.

3.2.2 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 applications. Section 3.3.1 discusses how we address this challenge.

3.2.3 Local Formula Generation

DDRI translates a program into a local-formula using Floyd-style verification condition generation [22]. The DDRI generator most resembles the generator employed by Proof-Carrying Code (PCC) [38] in that it works by symbolically executing the program [28] at the instruction level, and produces a formula representing execution along multiple program paths. However, because the PCC and DDRI generators address 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 sacrifices precision: it unrolls the loop a small but fixed number of times (similar to the unrolling done by ESC-Java [21]) and then uses Engler's underconstrained execution to fast-forward to the end of the loop. The number of unrolls is computed as the minimum of 100 and the number of iterations to the next recorded system event (e.g., syscall) as determined by its branch count. Unrolling the loop 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 reason about 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, DDRI models only those pages that may have been accessed in the original run by the symbolic dereference. DDRI knows what those pages are because DCR recorded their IDs in the original run using conventional page-protection techniques. In some instances, the number of potentially touched pages is large, in which case DDRI sacrifices soundness for the sake of efficiency: it considers only the subset of potentially touched pages referenced by the past k direct accesses.

3.3 Scaling Debugger Response Time

A primary goal of DCR is to provide responsive and interactive replay-debugging. But to achieve this goal, DCR's

inference method (DDRI–the post-run inference method introduced in Section 3.2) must surmount major scalability challenges.

3.3.1 Huge Formulas, NP-Hard Constraints

Modern constraint solvers cannot directly solve DDRI-generated formulas, for two reasons. First, the formula may be terabytes in size. This is not surprising as DDRI must reason about long-running data-processing code that handles terabytes of unrecorded data. Second, and more fundamentally, the generated formulas may contain NP-hard constraints. This too is not surprising as datacenter applications often invoke cryptographic routines (e.g., Hypertable uses MD5 to name internal files).

Just-in-Time Inference. 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 applications. 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, 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 [20].

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-DDRI 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.

3.3.2 Distributed System 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 ensuring causality in inferred datacenter runs is hard: it requires efficiently reasoning about communications spanning thousands of CPUs, possibly spread across thousands of nodes. JIT-DDRI can help with such reasoning by solving only those constraints involved in the chain of causality of interest to the developer. However, if the causal chain is long, then even JIT-DDRI-produced constraints may be overwhelmingly large for the solver.

Inter-Node Causality Relaxation. To overcome this challenge, DCR enables the user to limit the degree d of inter-node causality that it reasons about–a technique previously employed by the ODR system to scale multi-processor inference [7]. Specifically, if d is set to 0, then DCR does not guarantee any data-plane causality. That is, an inferred run may exhibit data-plane values received on one node that were never sent by another node. On the other hand, if d is set to 2, for instance, then DCR provides data-plane values consistent across

two node hops. After the third hop, causal relationships to previously traversed nodes may not be discernible.

The appropriate value of d depends on the system and error being debugged. We observe that, in many cases, reasoning about inter-node data-plane causality is altogether dispensable (i.e., $d = 0$). For example, figuring out why a lookup went to slave node 1 rather than slave node 2 requires tracing the causal path of the lookup request (a control-plane entity), but not that of the data being transferred to and from the slave nodes. In other cases, data-plane causality is needed–for example, to trace the source of data corruption to the underlying control-plane error on another node. We have found that if the data corruption has a short propagation distance, then $d \leq 3$ often suffices (see our case study in Section 5.2 for an example in which d had to be at least 2).

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 8 person-years. Here we present a selection of the implementation challenges we faced.

4.1 Sample Usage

With DCR, a user may record and replay-debug a distributed system with a few simple commands. Before starting a production recording, however, we wish to first configure DCR to never exceed a low threshold logging rate:

```
d0:~/$ dcr-conf Sys.MaxRecRate=100KBps
```

Next the user may start a recording session. Here we start Hypertable (a distributed database) on three production nodes under the "demo" session name:

```
p0:~/$ dcr-rec -s "demo" ht-lock-man
p1:~/$ dcr-rec -s "demo" ht-master
p2:~/$ dcr-rec -s "demo" ht-slave
```

Before initiating a replay debugging session, we use DCR's session manager to first identify the set of nodes that we wants to replay debug:

```
d0:~/$ dcr-sm --info "demo"
Session "demo" has 3 node(s):
[0]  p0 ht-lock-man 10m
[1]  p1 ht-master   32m
[2]  p2 ht-slave    11m
```

The output shows that though the master node ran for 32 minutes, the lock-manager and slave terminated early at about 10 minutes into execution. So we begin a replay-debugging session for just the early terminating nodes near the time they terminated:

```
d0:~/$ dcr-gdb --time 9m:12m
         --nodes 0,2 "demo"
gdb> backtrace node 0
#1 <segmentation fault>
#2 LockManager::handle_message():52
```

The output shows that node 0 terminated due to a segmentation fault, hence probably bringing down the slave sometime thereafter.

4.2 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.

4.3 Formula Generation

DDRI generates a formula by symbolically executing the target program (see Section 3.2.3), in manner very similar to that of the Catchconv symbolic execution tool [35]. 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.

4.4 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 [24]. 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

Here we present the experimental evaluation of DCR.

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 applications in our suite. Rather than compare apples with oranges, we base our comparison on a modified version of DCR, called BASE, that records both control and data plane non-determinism in a fashion most similar to SMP-ReVirt [18]–the state of the art in classical multi-core deterministic replay.

In short, we found that DCR incurs very low recording overheads suitable for at least brief periods of production use (a 16% average slowdown and 8 GB/day log rates). Moreover, we found that DCR's debugger response times, though sluggish, are generally fast enough to be useful. By contrast, BASE 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-like workloads.

5.1.1 Setup

Applications. We evaluate BASE and DCR on two real-world datacenter applications: *Cloudstore* [1] and *Hyptertable* [3].

Cloudstore is a distributed filesystem written in 40K lines of multithreaded C/C++ code. It consists of 3 sub-programs: the master server, slave server, and the client. The master program consists mostly of control-plane code: it maintains a mapping from files to locations and responds to file lookup requests from clients. The slaves and clients have some control-plane code, but mostly engage in control plane activities: the slaves 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 4 key sub-programs: the master server, metadata server, slave server, and client. The master and metadata servers are largely control-plane in nature–they coordinate the placement and distribution of database tables. The slaves store and serve the contents of tables placed there by clients, often without the involvement of the master or the metadata server. The slaves and clients are thus largely data-plane entities.

Workloads and Testbed. We chose the workloads to mimic peak datacenter operation and to finish in 20 minutes. Specifically, for *Hypertable*, 8 clients performed concurrent lookups and deletions to a 1 terabyte table of web data. Hypertable was configured to use 1 master server, 1 meta-data server, and 4 slave servers. For *Cloudstore*, we made 8 clients concurrently get and put 100 gigabyte files. We used 1 master server and 4 slave servers.

All applications were run on a 10 node cluster connected via Gigabit Ethernet. Each VM in our cluster operates at 2.0GHz and has 4GB of RAM. The OS used was Debian 5 with a 32-bit 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 Recording Overheads

Logging Rates. Figure 3 gives results for the record rate, a key performance metric for datacenter workloads. It shows that, across all applications, DCR's log rates are suitable for the datacenter–they're less than those of traditional console logs (100 KBps) and up to two orders of magnitude lower than BASE's rates (3 TB/day v. 8 GB/-day). This result is not surprising because, unlike BASE, DCR does not record data-plane I/O.

A key detail is that DCR outperforms BASE for only data-intensive programs such as the Hypertable slave nodes; control-plane dominant programs such as the

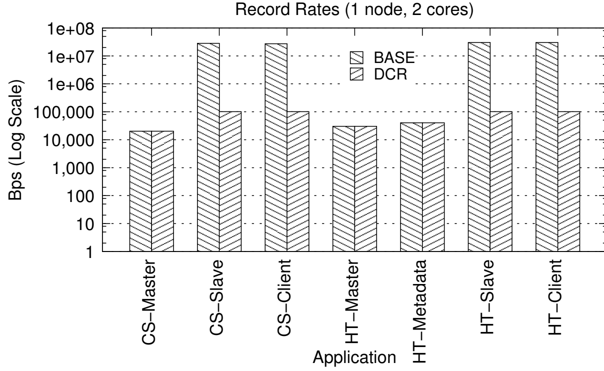

Figure 3: Per-node record rates for BASE and DCR. DCR's rate is up to two orders-of-magnitude lower because it logs just the control-plane I/O.

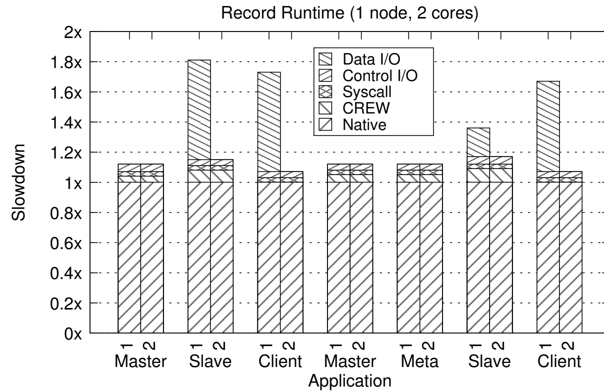

Figure 4: Record runtimes, normalized with native application execution time, for (1) BASE which records control and data planes and (2) DCR which records just the control plane. DCR's performance is up to 60% better.

Hypertable master node perform equally well on both. This makes sense, as data intensive programs routinely exceed DCR's 100 KBps logging rate threshold and are capped. The control-plane dominant programs never exceed this threshold, and thus all of their I/O is recorded.

Slowdown. Figure 4 gives the slowdown incurred by DCR broken down by various instrumentation costs. At about 17%, DCR's record-mode slowdown is as much as 65% less than BASE's. Since DCR records just the control-plane, it doesn't have to compete for disk bandwidth with the application as BASE must. The effect is most prominent for disk intensive applications such as the CloudStore slave and Hypertable client. 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 2%. The main bottleneck is shared-memory channel interpositioning, with CREW faults largely to blame–the Hypertable range servers can fault up to 8K times per second. The page fault rate can be reduced by lowering the default control-plane threshold of 10K faults/sec. DCR would then be more willing to deem high data-rate pages as part of the data-plane and stop intercepting accesses to them. But the penalty is more work for the inference mechanism.

5.1.3 Replay-Debugging Latency

Despite a formidable inference task, DCR's JIT debugger enables surprisingly responsive replay-debugging of real datacenter applications. To show this, we evaluate DCR's replay-debugging latency under two configurations: without and with Just-in-Time Inference (JITI) enabled.

For both configurations, we obtained the debugger latency using a script that simulates a manual replay-debugging session. The script makes 10K queries for state from the first 10 minutes of the replayed distributed execution. The queries are focused on exactly one node and may ask the debugger to print a backtrace, return a variable's value (chosen from the stack context indicated by the backtrace), or step forward n instructions on that node. Queries that take longer than 20 seconds are timed out.

Impact of Just-in-Time Inference. Figure 5 gives the average debugger latency, with and without the JITI optimization, for our application suite. It conveys two key results.

First, DCR provides native debugger latencies for data-unintensive nodes (e.g., the Hypertable and CloudStore master nodes), *regardless of whether JITI is enabled or not*. Data-unintensive nodes operate below the control-plane threshold data rate, hence enabling DCR to efficiently record all transactions on those channels. Since all information is recorded, there is no need to infer it and hence no need to generate a formula and solve it–hence the 0 formula sizes and solving times. The result is that, as with traditional replay systems, the user may begin replay debugging data-plane unintensive nodes immediately.

Second, DCR has surprisingly fast latencies for queries of data-intensive programs (e.g., Hypertable and CloudStore slaves), but *only if JITI is enabled*. Data-intensive programs operate above the control-plane threshold data rate and thus DCR does not record most of their I/O. The resulting inference task, however,

Program	(a) Without JITI Total	FormGen	FormSolve	(b) With JITI Total	FormGen	FormSolve
CloudStore						
Master	native	0 (0 GB)	0	native	0 (0 GB)	0
Slave	timeout	timeout (500 GB)	-	15	5 (270 KB)	10
Client	timeout	timeout (300 GB)	-	13	4 (90 KB)	9
Hypertable						
Master	native	0 (0 GB)	0	native	0 (0 GB)	0
Meta	native	0 (0 GB)	0	native	0 (0 GB)	0
Slave	timeout	timeout (500 GB)	-	16	5 (880 KB)	11
Client	timeout	timeout (200 GB)	-	13	4 (540 KB)	9

Figure 5: Mean per-query debugger latencies in seconds broken down into formula generation (FormGen) and solving time (FormSolve). Formula generation (FormGen) times out at 1 hour. The key result is that data-unintensive applications exhibit low latencies regardless of whether JITI is used or not, but data-intensive applications require JITI to avoid query timeouts.

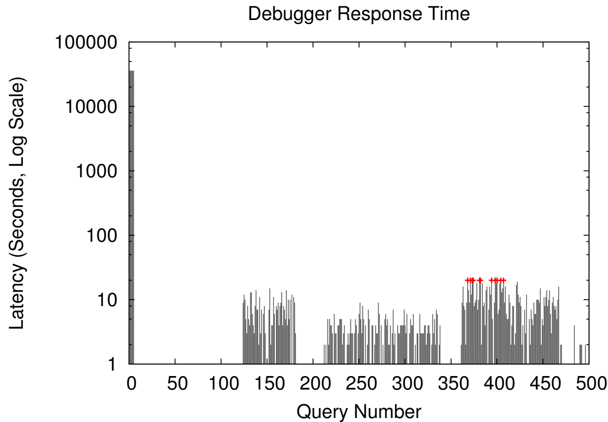

Figure 6: Debugger query latency profile for a Hypertable slave server. The first query is really slow, but subsequents ones are generally much faster. Red dots denote queries that timed out at 20 seconds.

is insurmountable without JITI, because a mammoth formula (often over 500 GBs) must be generated and solved. By contrast, JITI also produces large formulas. But these formulas are smaller (around 30 GBs) and are subsequently split into multiple smaller sub-formulas (500 KB on average) that can be solved fairly quickly (10 seconds on average).

User Experience. DCR's mean response time with JITI, though considerably better than without JITI, is still sluggish. Should the user expect every JITI query to take so long? The debugger latency profile given for a Hypertable slave node in Figure 6 answers this question in the negative. Specifically, it makes two points.

First, the slowest query by far is the very first query– it takes 10 hours to complete. This makes sense because the first query induces the replay engine to generate a multi-gigabyte formula and split it in preparation for Just-in-Time Inference. Though both of these operations take time linear in the length of the execution segment being debugged, they are slow when they have to process gigabytes of data.

The second key result is that non-initial queries are generally fast, with the exception of a few timeouts due to hard constraints (2% of queries in Figure 6). We attribute the speed to three factors.

First, results of the formula generation and splitting done in the first query are cached and reused in subsequent queries, hence precluding the need to symbolically execute and split the formula with each new query. Second, many queries (38% in Figure 6) are directed at concrete state (usually to control-plane state). These queries do not require constraint solving. Finally, if a query is directed at data-plane state, then DCR's debugger (with JITI) solves only the sub-formula corresponding to the queried state (see Section 3.3.1). These sub-formulas are generally small and simple enough (on the order of hundreds of KBs, see Figure 5) to provide 8-12 second response times.

5.2 Case Study

Here we report our experience using DCR to debug a real-world non-deterministic failure. We offer this experience not as conclusive evidence of DCR's utility–a difficult task given the variable amount of domain knowledge the developer brings to the debugging process– but as a sampling of the potential that DCR may fulfill with further study.

5.2.1 Setup

We focus our study on Hypertable issue 63–a critical defect entitled "Dropped Updates Under Concurrent Loading" [4]. Recent versions of Hypertable do not exhibit the issue, as it was fixed long ago. So we reverted to an older version [2] that did exhibit the issue.

Failure. Updates to a database table are lost when multiple Hypertable clients concurrently load rows into the same table. The load operation appears to be a success–clients nor the slaves receiving the updates produce error messages. However, subsequent dumps of the table don't return all rows; several thousand are missing.

Root Cause. In short, the data loss results from rows being committed to slave nodes (a.k.a, 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 keys in the dumped data. The committed keys are merely ignored.

The erroneous commits stem from a race condition in which row ranges migrate to other slave nodes while a recently received row within the migrated range is being committed to the current slave node. Instead of aborting the commit for the row and forwarding it to the newly designated data node along with other rows in the migrated range, the data node allows the commit to proceed.

5.2.2 Challenges and Lessons

We gained two key insights in the process of isolating the root cause:

Data-plane causality is sometimes necessary. We wanted to know if it was possible to debug this failure without data-plane causality, so we started the debugger with $d = 0$ (see Section 3.3.2). Surprisingly, our initial attempt to reproduce the failure was a success–we saw that several previously submitted updates were indeed missing. But when we tried to backtrack from the client to the sending slave node, we found that the sent updates had no correspondence with the received updates, making further backtracing difficult.

By contrast, the same experiment with d set to 2 yielded causally consistent results. We were able to comfortably backtrack the dumped key to the client that initially submitted it. The penalty for reasoning about inter-node causality, however, was a 10-fold increase in JIT debugger latency.

Data-plane determinism is dispensible. We wanted to reason about updates dropped in the original using the replay execution, as would be possible in a traditional replay system. But this was challenging, because as we quickly learned, there was no discernible correspondence between the orignal lost updates and the inferred lost updates. The was clear in retrospect: because the value of the updates needn't be any particular string for the underlying error to be triggered, `DCR` inferred an aribtrary string that happened to differ from that of the original.

We overcame this challenge by ignoring the originally dropped updates. Instead, we focused on tracing the dropped updates in the inferred replay run. Because the unerlying error was a control-plane defect, the discrepencies in key values between the original and the inferred mattered little in terms of isolating the root cause. Both exercised the same defective code.

5.2.3 Debugging in Detail

We isolated the root cause with a series of distributed invariant checks, each performed with the use of a global predicate [24].

Check 1: Received and Committed?

Predicate. Were all keys in the update successfully received and committed by the range servers? To answer this question, we created a global predicate that fires when any of the keys sent by a client fails to commit on the server end.

Result. The global predicate did *not* fire, hence indicating that all keys were indeed received and committed by their respective nodes.

Predicate operation. During replay, the predicate maintains a global mapping from each key sent by a client to the range server that committed the key. If all sent keys do not obtain a mapping by the end of execution, then the predicate fires.

To obtain the mapping, the predicate places two distributed breakpoints. The first breakpoint is placed on the client side `RangeLocator::set(key)` function, which is invoked every time a key is sent to a range server. When triggered, our predicate inserts the corresponding key into the map with a null value. The second breakpoint was set on the slave side `RangeServer::update(key)` function, which is invoked right before a key is committed. When triggered, the predicate inserts the committing node's id as the value for the respective key.

Check 2: Commited to the Right Place?

Predicate. The keys were committed, but were they committed to the right slave nodes? To find out, we created a global predicate that fires when a key is committed to the "wrong" node. The committing node is wrong if it is not the node reponsible for hosting the key, as indicated by Hypertable's global key-range to node-id table (known as the METADATA table).

Result. Partly through the execution our global predicate fired for row-key x. It fired because, although key x lies in a range that should be hosted by node 2, it was actually committed to node 1. Thus, some form of METADATA inconsistency is to blame.

Predicate operation. The predicate maintains two global mappings and fires when they mismatch. The first mapping maps from key-ranges to the node id responsible for hosting those key-ranges, as indicated by the METADATA table. The second maps each sent key to its committing node's id.

To obtain the first mapping, the predicate intercepts all updates to the METADATA table. This was done by placing a distributed breakpoint on the `TableMutator::set(key, value)` function. When the breakpoint fires and if `this` references a METADATA table, we then map `key` to `value`.

To obtain the second mapping, the predicate places a distributed breakpoint at the callsite of `Range::add(key)` within the `RangeServer::update(key)` function. When it fires, the predicate maps `key` to the committing node id.

Check 3: A Stale Mapping to Blame?

Predicate. We know that, before committing a key, a range server first checks that it is indeed responsible for the key's range. If not, the read/update is rejected. But Part 2 showed that the key is committted even after the range server's self check. Could it be the case that the node assignment for the committing key changed in between the range server's self check and commit? To find out, we created a global predicate that fires when a key's node assignment changes in the time between the self-check and commit.

Result. The predicate fired. The cause was a concurrent migration of the key range (known as a split) fielded by another thread on the same node. It turns out that range-servers split their key-ranges, offloading half of it to another range-server, when the table gets too large.

Predicate operation. This predicate maintains three mappings: (1) from keys to node ids at the time of the self check, (2) from keys to node ids at the time of commit, and (3) from keys to METADATA change events made in between the self check and commit. The predicate fires when there is an inconsistency between the first and the second.

To obtain the first and second mappings, we placed a breakpoint on calls to `TableInfo::find_containing_range(key)` and `Range::add(key)`, respectively. `find_containting_range()` checks that the `key` should be committed on this node and add() commits the row to the local store. When either of these breakpoints fire, the predicate adds a mapping from the `key` to the node id hosting that `key`. The predicate obtains the node id by monitoring changes to the METADATA table in the same manner as done in Part 2.

To obtain the third mapping, the predicate places a breakpoint on calls to `TableMutator::set(key)`, where the table being mutated is the metadata table.

	Always-on operation	Whole-system replay	Out-of-the-box use	Determinism
Hardware support [16, 32, 36], CoreDet [9]	*No*	*No*	*No*	Value
liblog [25], VMWare [5], PRES [39], ReSpec [33]	*No*	*No*	Yes	I/O
SherLog [44]	Yes	*No*	*No*	Output
ODR [7], ESD [45]	Yes	*No*	Yes	Output
Instant Replay [31], DejaVu [29]	Yes	*No*	Yes	Value
R2 [27]	Yes	Yes	*No*	Value
`DCR`	**Yes**	**Yes**	**Yes**	**Control**

Figure 7: A comparison with other replay-debugging systems. Only `DCR` meets all the requirements for *datacenter applications*.

6 Related Work

Figure 7 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 [40], PRES [39], and ReSpec [33]) and hardware and/or compiler assisted systems (e.g., Capo [36], Lee et al. [32], DMP [16], 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-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 [25], Friday [24], VMWare [5], Capo [36], PRES [39], and ReSpec [33] 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 [7] (our prior work), ESD [45], and SherLog [44] can efficiently replay some single-node applications (ESD more so than the others). But these systems were not designed for distributed operation, much less datacenter applications. Indeed, even single-node replay is a struggle for these systems. On long-running and sophisticated datacenter applications (e.g., JVM-based applications), 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 [9], ESD [45], and SherLog [44] 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, many datacenter applications run within the JVM, well-known for dynamically generating code.

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 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 have presented DCR, a replay debugging system for datacenter applications. We believe DCR is the first to provide 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 behavior of the control plane–the most error-prone component of the datacenter app. 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] Cloudstore. http://kosmosfs.sourceforge.net/.

[2] Git commit id of hypertable issue 63. 5e2045aeeb0f38db48d2eddf484b2d00445a1d59.

[3] Hypertable. http://www.hypertable.org/.

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

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

[6] ALTEKAR, G. An empirical study of the control and data planes. Tech. Rep. UCB/EECS-2010, EECS Department, University of California, Berkeley, 2010.

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

[8] ALTEKAR, G., AND STOICA, I. Dcr: Replay debugging for the data center. Tech. Rep. UCB/EECS-2009-108, EECS Department, University of California, Berkeley, May 2010.

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

[12] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *USENIX* (2004).

[13] 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).

[14] CROWLEY, P. *Network Processor Design: Issues and Practices*. 2002.

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

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

[17] DUNLAP, G. *Execution replay for intrusion analysis*. PhD thesis, Ann Arbor, MI, USA, 2006.

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

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

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

[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] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI* (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] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976), 385–394.

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

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

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

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

[33] 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).

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

[35] 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.

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

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

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

[39] 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).

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

[41] TANENBAUM, A. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

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

[43] 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).

[44] 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).

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