Automating the Debugging of Datacenter Applications with ADDA

Gautam Altekar
Cristian Zamfir
George Candea
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-22
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-22.html

April 4, 2011

Automating the Debugging of Datacenter Applications with ADDA

Gautam Altekar[1], Cristian Zamfir[2], George Candea[2], Ion Stoica[1]

UC Berkeley[1] and *EPFL, Switzerland*[2]

Abstract

Debugging data-intensive distributed applications running in a datacenter ("datacenter applications") is complex and time-consuming. Developers wish they had a way to deterministically replay failed executions with little human effort, but unfortunately no such tool exists today. We see two challenges in replay-based debugging: First, the clusters used to run datacenter applications consist of many nodes, so the *nondeterminism* resulting from multithreaded execution on a single node is compounded by the size of the cluster. Second, datacenter applications produce terabytes of intermediate data shipped from one node to the next—the total *data volume*, itself proportional to cluster size, makes full input recording for potential subsequent replay infeasible.

We present ADDA, a replay-debugging system for datacenter applications. We observe that these applications often consist of a separate "control plane" and "data plane," and that the applications' initial inputs are typically persisted in append-only storage for reasons unrelated to debugging. Building upon these observations, ADDA leverages the control / data plane separation to make recording of debug-critical data scalable even in large clusters, it deterministically re-synthesizes intermediate data based on the (already available) initial inputs, and performs reduced-scale replay, i.e., recreates failed executions on just a subset of the original cluster.

We show that ADDA scales well and deterministically replays real-world failures in Hypertable and Memcached. We also argue that ADDA's techniques generalize to a broader set of datacenter applications.

1 Introduction

More and more applications that we use on a daily basis, such as Web search, e-mail, social networks (e.g., Facebook, Twitter), and video sharing are hosted in the cloud. Furthermore, many businesses either use cloud-based services such as Salesforce and Google Docs, or deploy their applications in private clouds. These services often use cluster computing frameworks such as MapReduce, BigTable, Memcached and Cassandra that run on commodity-hardware clusters consisting of as many as thousands of machines. As users and businesses are growing more dependent on these hosted services, the frameworks and the application they use need to be highly robust and available. To maintain high availability, it is critical to diagnose the failures, and quickly debug these applications.

Unfortunately, debugging datacenter applications is hard. When an application failure occurs, the causality chain of the failure is often difficult to trace, as it may span multiple nodes. Furthermore, such applications typically operate on many terabytes of data every day and are required to maintain high throughput, which makes it hard to record what they do. Finally, these applications are usually part of complex software stacks that are used to provide 24×7 services, thus taking the application down for debugging is not an option.

A system-wide replay-based solution is the natural option for debugging, as it offers developers the global view of the application "on a platter:" by replaying deterministically a failure, one can use a debugger to zoom in on various parts of the system and understand why the failure occurs. If a system-wide replay was not possible, the developer would have to reason about global (i.e., distributed) invariants, which in turn can only be correctly evaluated at consistent snapshots in the distributed execution. Getting such consistent snapshots requires either a global clock (which is non-existent in clusters of commodity hardware) or complex algorithms to capture consistent snapshots, such as Chandy-Lamport [10].

However, developing an automated solution for replay-based debugging is harder for datacenter applications than for a single node, due to the inherent runtime overheads required to do record-replay. First, these applications are typically data-intensive, as the volume of data they need to process increases proportionally with the size of the system (i.e., its capacity), the power of individual nodes (i.e., more cores means more data flowing through), and ultimately with the success of the business. Recording such large volumes of data is impractical. A second reason is the abundance of sources of non-determinism. Coordinating cluster nodes to perform a faithful replay of a failed execution requires having captured all critical causal dependencies between control messages exchanged during production. Knowing a priori which dependencies matter is undecidable. A third challenge is of a non-technical nature: as economies

of scale are driving the adoption of commodity clusters, the tolerable runtime overhead actually decreases—making up for a 50% throughput drop requires provisioning twice more machines; 50% of a 10,000-node cluster is a lot more expensive than 50% of a 100-node cluster. When operating large clusters, it actually becomes cheaper to hire more engineers to debug problems than to buy more machines to tolerate the runtime overhead resulting from an automated record-replay system.

Existing work in distributed system debugging does not offer solutions to these challenges. Systems like Friday [15] address distributed replay, but do not address the data intensive aspect, therefore they are not suitable for data centers. No existing system can replay at this scale and magnitude. If we are to cast off our continued reliance on humans to solve routine debugging tasks, we must devise smarter, more scalable debugging tools that aid developers in quickly debugging problems.

To address these challenges, we propose ADDA (*a*utomated *d*ebugger for *d*ata center *a*pplications), a replay-based debugging system. Three insights make ADDA practical. First, data center applications are almost always split into a control plane and a data plane, and most bugs reside in the former [6], so a deterministic replay of the control plane enables the debugging of most problems. Second, inputs that enter data center applications are typically persisted in append-only storage (e.g., for compliance, fault tolerance) and thus retrievable at the time of debugging. When combined with the ability to replay the control plane, this property allows ADDA to do away with recording all inputs. Third, with suitable recording, it is possible to deterministically synthesize all intermediate data sets during debugging, thus voiding the need to record intermediate data.

In this paper, we make three contributions:

- A scalable technique for *recording the behavior* of datacenter applications with low overhead;
- A practical technique for *synthesizing intermediate data* to enable replay-based debugging, in a way that is not affected by nondeterminism;
- A technique we call *reduced-scale replay*, which allows developers to replay a failed execution that occurred in the production cluster on a different smaller cluster or on a partition of the original cluster.

In the rest of this paper, we provide background necessary to understand the problem and our ensuing design (§2), we present ADDA's design (§3), our ADDA prototype (§4), we evaluate ADDA (§5), present related work (§6), future work (§7), and conclude (§8).

2 Overview

Many replay debugging systems have been built over the years and experience indicates that they are invaluable in reasoning about nondeterministic failures [16, 15, 20, 9, 4, 14, 22, 19, 24, 21]. However, we believe no existing system meets the demands of the datacenter environment. We discuss these requirements next.

2.1 Design Requirements

Debug Determinism: To be useful, a replay-debugging system must be able to reproduce a production failure along with its root cause, which we refer to as debug determinism. It should be able to do this for most failures that occur in production, but it need not reproduce absolutely all failures to be considered useful (e.g., faithful reproduction of control-plane logic is often sufficient for datacenter systems [6]).

Whole-System Replay: The system should be able to replay the behavior of *all nodes* in the distributed system, if desired, after a failure is observed. While this is doable on a small set of nodes, when scaling to large datacenters it becomes 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. Furthermore, every layer of the stack has to be replayed—for instance, merely using network sniffing tools like tcpdump and tcpreplay is insufficient to construct the global view that is required to understand what happened at the system level. Even if logging all network messages was feasible (which is not the case), we must account for the fact that nodes themselves may be nondeterministic, so merely replaying network messages is insufficient.

Low Record Overhead: Distributed applications often run on clusters consisting of hundreds or thousands of machines. In such large systems, even a moderate recording overhead can translate into significant operation and capital costs. Thus, achieving a low record overhead should be a major goal of any replay-debugging system for datacenter applications, even more so than for single machine applications.

Decoupled Debugging / Availability Concerns: Improving the debuggability of applications should not hurt service availability, especially for 24×7 services. This means that, upon failure, the operator's main concern should be to bring the system back up, not to keep the system in a state that will enable developers to debug the problem. Furthermore, runtime overhead must be reasonable, and ADDA should not increase the likelihood of

failure or lockups. Finally, it is not reasonable to expect the developers to either be permitted to replay a failure on the production cluster, or have access to an identical cluster as the production cluster. This leaves the developers with the option of *reduced-scale replaying*, i.e., replaying an entire cluster on a small set of development nodes.

Minimal Environment Assumptions: A replay-debugging system should be capable to 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 no modifications to the applications themselves. The reality of cluster-based services is that many components must be treated as black boxes (either because there is no source code available or they are too complicated), but still need to be replayable. Special languages and source-code modifications (e.g., custom APIs and annotations, as used in R2 [17]) are undesirable because they are cumbersome to learn, maintain, and retrofit onto existing datacenter applications. Source-code analysis is often prohibitive because some of the components may be closed-source. Finally, datacenter applications operate in a "mixed world": while the nodes running the application can be assumed to be traced using our instrumentation, other nodes (e.g., those running DNS or LDAP servers) may not be.

2.2 Insights Enabling Our Solution

In this section, we describe the main insights behind our replay-debugging system, ADDA.

2.2.1 Control-Plane Determinism Suffices

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

The control plane of a datacenter application is the code that manages or controls data-flow and implements operations like 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 99% of the newly-written code in datacenter software [12] and serve, not surprisingly, as breeding grounds for distributed data race bugs. On the other hand, the control plane accounts for only 1% of all datacenter traffic [6].

In contrast, datacenter application debugging rarely requires reproducing the same *data-plane* behavior. The data plane 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 1% of the code in a datacenter application [12] and are often part of well-tested libraries. Yet, the data plane is responsible for generating and processing over 99% of datacenter traffic [6].

Hence, by recording and replaying the control plane, ADDA *can reproduce most bugs and provide low overhead recording.*

2.2.2 Data-Plane Inputs Are Persistently Stored

The data that enters the system from outside is typically stored persistently in append only file systems such as GFS and HDFS. Thus, we can assume that these external inputs are readily available to developers when debugging the system. As a result, ADDA does not need to record these external inputs. *This is a crucial property of datacenter applications, as it saves us from recording prohibitive amounts of data.*

3 Design

In this section we present our approach. Then we discuss the key design challenges of efficiently recording and replaying datacenter applications, and we describe how ADDA addresses them.

3.1 Approach

The complex yet low data-rate nature of the control-plane motivates ADDA's *approach of relaxing its determinism guarantees*. Specifically, ADDA 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 ADDA to efficiently record all nodes in the system.

Our system architecture is given in Figure 1. Like other replay systems, it operates in two phases: record-mode and replay-mode.

3.1.1 Record Mode

What ADDA records. All ADDA-enabled nodes record *control plane nondeterminism*, by which we mean the ordering and content of control plane inputs and outputs (I/O). We consider thread scheduling order and asynchronous control-flow changes (e.g., signals and preemptions) to be part of the application control plane,

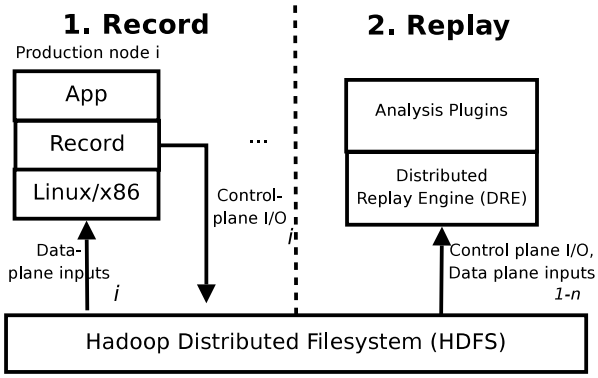

Figure 1: ADDA's distributed architecture and operation. It uses the recorded control-plane I/O and persistently stored data-plane inputs to generate, *in a best effort fashion*, a control-plane deterministic run.

and hence they are also recorded. Control plane non-determinism is recorded by all nodes regardless of whether the control plane I/O originated externally (i.e., from an untraced node) or internally (i.e., from an ADDA-traced node).

What ADDA does not record. ADDA-enabled nodes do *not* record data-plane I/O, regardless of whether the data plane I/O is external or internal. ADDA assumes that external data-plane inputs are stored persistently and available for use during replay. ADDA does *not* assume that internal data-plane I/O is stored persistently. Instead, it attempts to regenerate it in replay mode.

3.1.2 Replay Mode

ADDA's Distributed Replay Engine (DRE) uses the recorded control-plane I/O and the persistently-stored data-plane input to generate a control-plane deterministic run. The replay is best effort: while the DRE guarantees replay of control-plane nodes (the most complex and bug-prone components), it may not be able to replay multi-processor intensive data-plane nodes (the least complex and relatively bug-free component). Toward ADDA's goal of best-effort replay, the DRE was designed with the following key principles in mind:

Synthesize missing non-determinism when possible. While recording control plane non-determinism is sufficient for replaying control plane nodes in a distributed application, mixed control/data plane nodes (e.g., Hypertable's range server or a Hadoop tasktracker) require data-plane nondeterminism to deterministically replay even their control-plane components. The DRE attempts to recompute this unrecorded non-determinism in a best-effort fashion using Data Plane Synthesis (Section 3.3).

Provide a platform for automated debugging. Going beyond replay, the DRE also serves as a platform for writing powerful replay-mode analysis plugins. Namely, it exports an easy-to-program, Python-based plugin architecture for developing sophisticated distributed analyses. In fact, we've used it to write several distributed analyses such as distributed data flow and global invariant checking (Section 3.4).

3.2 Recording Control Plane Non-determinism

To record control-plane non-determinism, ADDA must first identify it. *In the general case*, manually identifying is hard – it usually requires a deep understanding of program semantics, and in particular, whether or not the nondeterminism emanates from control-plane code.

The key observation behind ADDA is that, in its target domain of datacenter applications, the control plane can often be manually identified with ease, and if not, automatic methods can be successfully applied. This observation motivates ADDA's approach of semi-automatically classifying control-plane nondeterminism. In particular, ADDA interposes on communication channels (Section 3.2.1) and then records the ordering and values from only those channels that are semi-automatically classified as control-plane channels (Section 3.2.2).

3.2.1 Interposing on Channels

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

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

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

To efficiently detect inter-CPU sharing, ADDA employs the page-based Concurrent-Read Exclusive-Write (CREW) memory sharing protocol, first suggested in the context of deterministic replay by Instant Replay [18] and later implemented and refined by SMP-ReVirt [14]. Page-based CREW leverages page-protection hardware found in modern MMUs to detect concurrent and conflicting accesses to shared pages. When a shared page comes into conflict, CREW then forces the conflicting CPUs to access the page one at a time, effectively simulating a synchronized communication channel through the shared page. Details of our CREW implementation are given in Section 4.2.1.

3.2.2 Classifying Channels

Two observations underly ADDA's semi-automated classification method. The first is that, for datacenter applications, control plane channels are easily identified. For example, Hypertable's master and lock server are entirely control plane nodes by design, and thus all their channels are control plane channels. The second observation is that control-plane channels, though bursty, operate at low data-rates [6]. For example, Hadoop job nodes see little communication since they are mostly responsible for job assignment – a relatively infrequent operation.

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

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

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

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

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

3.2.3 Taming False Sharing with Best-Effort CREW

The CREW protocol, under certain workloads, can incur high page-fault rates than in turn will seriously degrade performance (see SMP-ReVirt [13]). Often this is due to legitimate sharing between CPUs, such as when CPUs contend for a spin-lock. More often, however, the sharing is 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, ADDA employs a simple strategy to avoid high page-fault rates. When ADDA observes that the fault-rate results in token-bucket overflow (suggesting that the page 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. If not, CREW is disabled once again, and the cycle repeats. When CREW is selectively disabled, we can still provide replay, but only if the data-race free assumption is met for those pages (ADDA records the lock order to handle this case).

3.3 Providing Control Plane Determinism

The central challenge faced by ADDA's Distributed Replay Engine (DRE) is that of providing a control-plane deterministic view of program state. This is challenging because, although ADDA knows the original control-plane inputs, it does not know the original data-plane inputs. Without the data-plane inputs, ADDA 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 a novel technique we call Data-Plane Synthesis (DPS). The key observation behind DPS is that external data plane inputs are persistently stored by the application and are avaialbe for use during record time. Click-logs, for example, are stored in HDFS by default in case they are processed incorrectly and the processing needs to be redone. This observation then motivates our approach of using the stored data plane inputs to regenerate the communication on data plane channels. DPS is essential as it enables ADDA to synthesize data plane inputs without resorting to expensive inference techniques.

3.3.1 Regenerating Intermediate Inputs

The external data-plane inputs can be used to replay those processes that read it directly (i.e., front-end systems). However, a transformed version of those inputs is typically passed to internal/intermediate nodes in the traced datacenter application by the front end system. Consequently, ADDA cannot simply use the external data-plane inputs to replay these intermediate nodes.

To address this challenge, we employ the classic technique of order-based replay [18] to regenerate the content of intermediate data-plane inputs. The observation underling order-based replay is that, given the original inputs to a computation, and the ordering of channel communications on a node, one can deterministically reproduce the original outputs of that node.

The key challenge that ADDA must address is that of applying order-based replay to all internal/recorded nodes. We describe how ADDA provides order-based replay with a short inductive proof. We being with the base case and then employ the inductive step with two nodes.

Base Case. The base case is that of replaying the data-plane outputs of a single node, given access to all inputs (either because it was logged or was persistent external data-plane input). If shared memory interleavings on the node are replayed, then it will generate the same data-plane outputs.

Inductive Step. It suffices to show that given two order-replayed nodes A and B, B will receive the same inputs (and hence generate the same outputs) if B reads messages from A using the original ordering and message size, blocking if necessary.

3.3.2 Dealing with a Mixed World

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

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

First, replaying applications will need to obtain data plane inputs from the store during replay, but these original inputs may no longer be present on the same nodes at replay time. HDFS, for instance, may redistribute blocks or even alter block IDs. Hence simply reissuing HDFS requests with original block IDs is inadequate. The second challenge is that our target application may use the persistent store to hold temporary files: Hadoop, for instance, stores the results of map jobs to temporary files within an HDFS cluster. Since data read from these files are part of the data-plane, ADDA will not have recorded them. Moreover, DPS cannot synthesize them because HDFS is not recorded/replayed.

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

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

First, nodes may be replayed on hosts different than those used in the original, making it hard for DPS to

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

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

REPLAYNET poses two key challenges:

Obtaining Unique Message IDs. To send and receive messages on REPLAYNET, senders and receivers must be able to identify messages with unique IDs. These message IDs are simple UUIDs that are assigned at record time. Conceptually, the message ID for each message is logged by both the sender and receiver. The receiver is able to record the message ID since the sender piggy-backs it (using an in-band technique we developed in the liblog system [16]) on the outgoing message at record time. Further details of piggy-backing are given in Section 4.2.3.

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

3.3.3 Coping with Unrecorded Shared-Memory Ordering

A key requirement of order-based replay is that complete ordering information must be available to guarantee replay. Unfortunately, ADDA's recording of shared memory interleavings may be incomplete since it disables CREW for high data-rate pages (Section 3.2.3). In particular, the interleaving of data races on such pages will not be recorded, hence precluding the reproducion of computation on intermediate data-plane inputs and the subsequently generated outputs. *Hence* ADDA *does not guarantee replay of mixed control/data-plane nodes in multiprocessors.*

Despite this limitation, ADDA ensures that control plane exclusive components (e.g., Hypertable's master or lock server) can always be replayed independently of whether data-plane nodes can be replayed or not. This holds for two reasons. First all inputs on control plane nodes are recorded, because all such inputs are control plane in nature. Second, shared-memory data rates on control plane nodes are, in our experience, extremely low, and therefore ADDA is able to capture all CREW ordering information. We argue that the ability to replay control plane nodes is still valuable because the control plane accounts for most bugs in the distributed system [6].

3.4 Enabling Automated Debugging

A replay system in of itself isn't particularly useful for debugging. That's why ADDA goes beyond replay to provide a powerful platform for building powerful replay-mode, automated debugging tools. In particular, ADDA was designed to be extended via plugins, hence enabling developers to write novel, sophisticated, and heavy-weight distributed analyses that would be too expensive to run in production. We've created several plugins using this architecture, including distributed data flow, global invariant checking, communication graph analysis, and distributed-system visualization. Figure 2 shows the visualization plugin in action on a replay of the Mesos cluster operating system.

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

3.4.1 Plugin Programming Model

ADDA plugins are written in the Python programming language, which we selected for its ease of use and suitability for rapid prototyping. A key goal of ADDA's plugin architecture is to ease the development of sophisticated plugins. Toward this goal, ADDA plugin model provides the following properties, many of which are

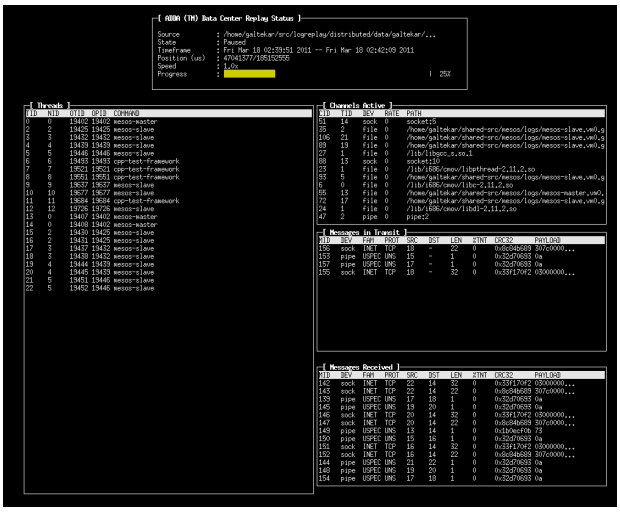

Figure 2: An ADDA visualization plugin running on a replay of the Mesos cluster operating system. This ncurses-based plugin provides a dynamic, birds-eye view of the distributed system in replay, including active threads/nodes, open communication channels (sockets, pipes, files), and in-transit/received messages.

borrowed from the Friday distributed replay system [15]:

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

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

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

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

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

Access to fine-grained analysis primitives. Plugin developers shouldn't have to reinvent the wheel. Thus, ADDA comes preloaded with commonly-used, powerful, fine-grained analysis primitives. An example of such a primitive is ADDA's data-flow analysis primitive. The primitive exports two key functions (is_tainted(node, addr) and set_taint(node, addr)) that plugins can invoke to determine if some origin data has influenced and to taint the byte at addr, respectively. Other primitives include lock-tracing, race-detection, and formula generation (whereby one may obtain the precise logical relationships between between program state). These primitives require introspecting replay execution at instruction level, which ADDA does using binary translation (discussed in more detail in Section 4.3.2).

3.4.2 Example Plugin : Distributed Data Flow

Here we focus on the design of DDFLOW—a distributed data-flow analysis plugin we've built using the ADDA toolkit. DDFLOW's goal is to provide a trace of all instructions or functions that operate, transitively, on the contents of a (user-specified) origin data file or message. DDFLOW is particularly useful in the context of data-loss bugs: it lets you track the flow of your data and helps you quickly identify where your data finally ends up – a process that could take hours of manual searching if done manually. DDFLOW highlights the power of ADDA plugins because it is an example of a heavyweight analysis that can most easily be done during replay (in-production datacenter apps will timeout due to the overhead of such heavyweight analysis).

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

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

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

4 Implementation

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

4.1 Usage

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

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

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

To replay using the DDFLOW analysis plugin (see Section 3.4.2), one need only specify the plugin name and the location of previously collected recordings:

```
$ adda-replay --plugin=dtaint
    hdfs://i1/demo/*
```

4.2 Lightweight Recording of User-Level Code

A key observation behind ADDA's implementation is that bugs in datacenter applications often originate from within application code rather than from kernel code. After all, datacenter applications rarely involve kernel changes. This observation motivates ADDA's approach of tracing only the non-determinism needed to replay user-level code of developer-selected application processes.

4.2.1 Interpositioning

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

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

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

CREW in detail. Conceptually, ADDA's CREW implementation closely follows that of SMP-ReVirt's [14]: it maintains shadow page tables whose permission are upgraded and downgraded at CREW events. But unlike SMP-ReVirt, ADDA maintains shadow page tables only for those processes that are traced. Moreover, ADDA does not shadow kernel pages (they are identical to those in guest page tables) hence avoiding false sharing in the kernel (a significant bottleneck in SMP-ReVirt). ADDA interposes on page table operations using Linux's `paravirt_ops` interface in the same manner as Xen.

4.2.2 Asynchronous Events

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

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

To address this problem, ADDA leverages standard asynchronous even replay technique. These technique rely on the following two observations. First, one can precisely identify a point in program execution via the x86 triple <eip, ecx, branch count>. The second is that one can efficiently obtain the branch count from the hardware performance counters found in modern commodity machines.

4.2.3 Piggy-backing

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

ADDA employs two techniques, both of which leverage the semantics of system calls, to reduce piggy-backing overheads: *message-level piggy-backing* and *TCP-aware unique ids*. The key observation behind message-level piggy-backing is that data plane applica-

tions send data in large message chunks: Memcached, for instance, performs sys_sends on 2 MB buffers. ADDA leverages this observation by piggy-backing at the message level rather than the packet level.

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

4.3 Distributed Replay and Analysis

4.3.1 Serial Replay

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

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

4.3.2 Fine-Grained Analyses

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

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

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

5 Evaluation

In this section we aim to answer the following questions: a) Is ADDA effective in debugging real-world problems occurring in real-world data center applications? (§5.1) b) Is ADDA's recording overhead tolerable, and how does it scale with cluster size, input data volume? (§5.2) c) Is ADDA efficient in replaying failed executions for debugging? (§5.3).

5.1 Experience

In this section we describe how we used ADDA to succesfully reproduce and debug bugs in Hypertable [1]. Hypertable [1] is an open source high performance data storage designed for large-scale data-intensive tasks and is modeled after Google's Bigtable [11]. Hypertable is deployed at Baidu, the leading search services in China and the Rediff online news provider.

5.1.1 Data Loss in Hypertable

We used ADDA to debug a previously-solved Hypertable defect [2] that causes updates to a database table to be lost when multiple Hypertable clients concurrently load rows into the same table. This bug is hard to reproduce and its root cause spans across multiple nodes. The load operation appears to be a success—neither clients nor slaves receiving the updates produce error messages. However, subsequent dumps of the table do not return all rows—several thousand are missing. The data loss results from rows being committed to slave nodes (i.e., Hypertable range servers) that are not responsible for hosting them. The slaves honor subsequent requests for table dumps, but do not include the mistakenly committed rows in the dumped data. The committed rows are merely ignored. The erroneous commits stem from a race condition in which row ranges migrate to other slave nodes at the same time that a recently received row

within the migrated range is being committed to the current slave node.

Reproducing this failure required 8 concurrent clients that insert 500MB data into the same table, after which they check the consistency of the table. We recorded several executions with ADDA until the failure was reproduced—the recording overhead was similar to the one in Figure 6. Afterwards, we replayed the failure with ADDA in our development single-machine setup. We inserted breakpoints during row range migration, where we suspected the root-cause is located and we observed the data race occurring deterministically. ADDA's ability to reliably replay the failure combined with the bird's eye view of the entire system made debugging substantially easier and faster.

5.1.2 Hypertable Hang Under Memory Pressure

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

5.2 Recording Efficiency

We ran all experiments in a cluster with 14 machines with 2 Intel Xeon 3.06GHz processors, 2GB of RAM, two 7200RPM drives in RAID 0, running 32-bit Linux 2.6.29. The machines are in a single rack, have 1Gbps NICs, and are interconnected by a single 1Gbps switch.

The size of the cluster may not be representative of the size of current data centers, however, we used the largest cluster that was available to us and in which we had access to the bare-metal hardware. We could not use a virtualized environment such as EC2 because we needed access to the hardware branch counter in order to replay asynchronous events (§4.2.2).

We measured ADDA's recording performance versus the slowdown of the naive approach that records all inputs, in order to show the benefits of REPLAYNET. To simulate the naive approach, we configured ADDA to log all inputs. We first evaluate the single processor case (§5.2.1), then the logging overhead (§5.2.2) and then the multiple CPU case (§5.2.3).

5.2.1 Runtime Overhead

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

Memcached Memcached [3] is a high-performance, distributed memory object caching system, typically used for speeding dynamic web applications by alleviating database load. Memcached is used by online services providers such as Youtube, Wikipedia, Flickr, etc.

To evaluate the efficiency of recording a memcached deployment, we simulated a photography blog Web application in which memcached is used by the user-facing Web application server to cache the files containing the photos. This setup resembles the Facebook photo storage [7], in which memcached is used to reduce latency. We assume that the photos are stored in persistent storage (i.e., HDFS) and the clients copy them from persistent storage to the memcached servers. We used various setups with a varying number of memcached servers, number of clients, and total input sizes. Each server and client run on separate machines. Each client randomly selects one of the memcached servers to either write or read a photo—reads are selected with 90% probability since reads are predominant in Facebook's daily photo traffic [7].

ADDA's recording overhead with varying size of the input from persistent storage (Figure 3) is between 18% and 23%. On the other hand, the naive approach imposes a high overhead: between 100% and 125%. This shows the benefits of REPLAYNET: logging all inputs causes the naive approach to have up to 5 times higher runtime overhead than ADDA.

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

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

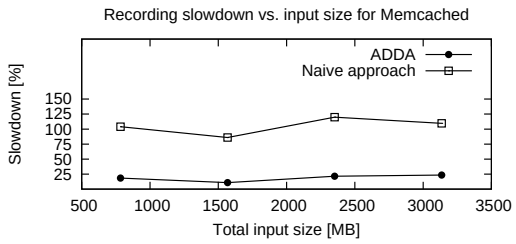

Figure 3: Recording slowdown in Memcached while varying the total size of the input from persistent storage.

This experiment shows that ADDA's overhead is between 20% and 65% and scales well with the number of nodes in the system. Moreover, ADDA scales well when the servers operate under heavy load. The naive approach has high overhead (up to 250%). However, as the memcached servers become saturated, clients become less loaded. Since in the naive approach clients are responsible for most of the logging (the workload is dominated by reads, which are fully recorded by clients), the impact of heavy logging for the naive approach decreases.

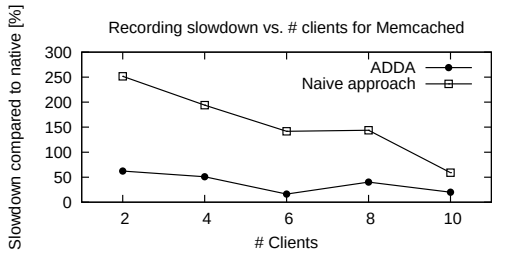

Figure 4: Recording slowdown for Memcached with varying number of clients.

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

The range servers store the content of the database tables in memory and also dump them to a distributed file system such as HDFS. Because ADDA currently requires that the target application uses a VFS-like interface to communicate with the file system (§3.3.2) we used a dedicated machine in our cluster as a dedicated shared file system for the range servers. In future work we intend to use the HDFS Fuse support and fix a bug in Hypertable that prevented us from experimenting with this setup.

Figure 5 shows that for Hypertable, the recording overhead scales well with the size of the input from persistent storage. The overhead, measured as transaction throughput, is in between 10% and 50%. On the other hand, the naive approach has higher overhead, which increases up to 90% for the largest total input size. In this experiment, Hypertable was configured with one master, one lock server, 3 range servers, and 7 clients that placed a heavy load on the system. Each client used an input file ranging from 30MB to 150MB. Clients read the input file from persistent storage.

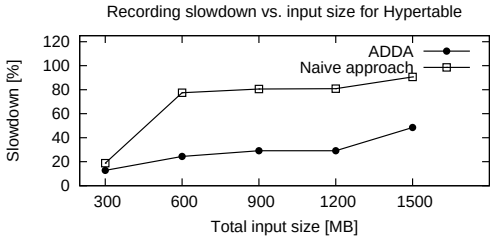

Figure 5: Recording slowdown for Hypertable with varying size of the input from persistent storage.

Figure 6 shows that ADDA scales well with the number of recorded nodes and the overhead is in between 40% and 50%. Due to higher logging rates, the naive approach has higher overhead. In this experiment, Hypertable was configured with one master, one lock server, 2 range servers, and a number of clients ranging from 3 to 9. Each component was run on a separate machine. The overhead is measured in terms of throughput loss.

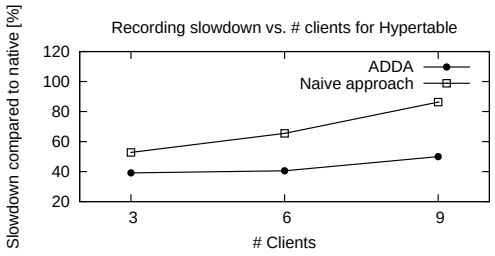

Figure 6: Recording slowdown in Hypertable with varying number of clients.

5.2.2 Log Size

ADDA has low logging rates. Figure 3 shows ADDA's log size for a memcached workload, while varying the total input size read from persistent storage. The naive approach also records internal inputs, therefore it produces an order of magnitude larger logs. For both systems the log size increases linearly with the input size, yet the slope is larger for the naive approach. Moreover,

memcached is designed so that server instances do not communicate with each other. If this would have been the case, we would expect that the log size for the naive approach to increase even more, due to the communication between memcached servers, while ADDA does not record this communication.

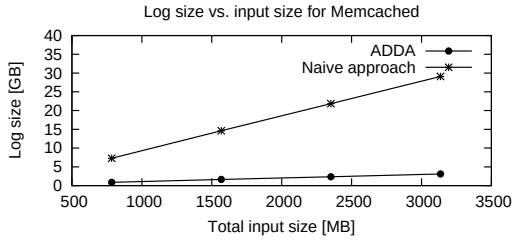

Figure 7: Log size for recording a memcached workload with varying input size from persistent storage. ADDA has 10 times smaller logs compared to the naive approach.

Hypertable exhibits a similar behavior (Figure 8). We expect these results to improve even more with a simple optimization: our current REPLAYNET prototype allocates a static 15KB entry for recording the meta-data associated with an I/O system call. However, this is typically too large: for Hypertable, log entries are dominated by zeros, which we could compress to 100X smaller size. By adding support for variable entry sizes, we expect ADDA's logging rates to improve substantially.

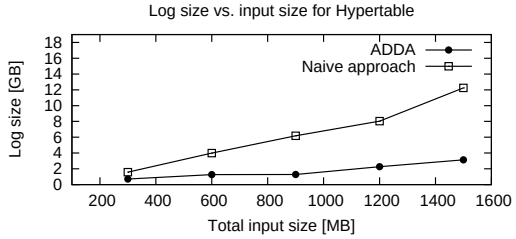

Figure 8: Log size for recording a Hypertable workload while varying the input size from persistent storage.

5.2.3 Performance for Multi-Processors

To validate our assumption about applying CREW selectively to control plane components such as the Hypertable master and lock server, we enabled CREW in the experiment in Figure 6. Thus, the control plane components were allowed to take advantage of both CPUs of their machines. In both cases, ADDA had the same slowdown compared to the baseline, showing that using CREW for the control plane components in Hypertable does not slow down the execution even when the system is under heavy load. This is confirmed by the small rate of CREW faults (at most 150 faults / sec) for each of these components.

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

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

5.3 Replay

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

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

6 Related Work

Classical single node replay systems such as Instant Replay [18], VMWare [4], and SMP-ReVirt [14] may be 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 distributed file system). Moreover, systems like WiDS [20] and Friday [15] provide distributed replay but do not address the data-intensive aspect.

Several relaxed-deterministic replay systems (e.g., PRES [22] and ReSpec [19]) and hardware and/or compiler assisted systems (e.g., Capo [21], CoreDet [8]) 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 such as datacenter systems.

R2 [17] 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 inputs, thus incurring low recording overheads. However, such annotations require considerable developer effort to manually identify the components that need to be recorded. On the other hand, ADDA makes this selection automatically.

Recent replay-debugging systems such as SherLog [23], ODR [5], and ESD [24] can efficiently replay some single-node applications while recording very little information, or no information at all. These systems use inference to recompute the missing runtime information. However, these systems were not designed for distributed operation, much less datacenter applications. Even for single node replay, these systems have to reason about an exponential number of program paths, which limits their ability to replay at the scale of the data center.

7 Future Work

Our on-going short term implementation goals are to test ADDA with other popular data center applications, such as Hadoop and Cassandra. Moreover, we are working on parallel replay. This can be done by allowing nodes to proceed in parallel during replay and enforce the relative order given by the Lamport clocks that ADDA already records.

ADDA supports multi-processor recording, but does not yet support multi-processor replay. We plan to add this in future work. Replaying CREW events is not challenging in itself if all CREW events are recorded. However, if ADDA would use control plane code selectivity for CREW recording, it may miss some important CREW events, therefore replay may not be possible. In this case, a solution can be to use the inference approaches used in PRES [22] or ODR [5], at the expense of larger replay times.

8 Conclusion

In this paper we presented ADDA, a replay-debugging system for datacenter applications. To achieve low overhead recording, ADDA leverages the "control plane" and "data plane," separation typical in these applications, as well as the availability of the data plane inputs in persistent storage . We show that ADDA scales well, has low overhead and logging rates, and deterministically replays real-world failures in popular data center applications.

References

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

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

[3] memcached. http://www.memcached.org/.

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

[5] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore programs. In *SOSP*, 2009.

[6] G. Altekar and I. Stoica. Focus replay debugging effort on the control plane. In *HotDep*, 2010.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. In *OSDI*, 2010.

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

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

[10] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1), 1985.

[11] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[12] P. Crowley. *Network Processor Design: Issues and Practices*. 2002.

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

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

[15] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. 2007.

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

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

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

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

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

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

[22] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. Do you have to reproduce the bug at the first replay attempt? – PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[23] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*, 2010.

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