Automating the Debugging of Datacenter Applications with ADDA

Cristian Zamfir* · Gautam Altekar† · Ion Stoica† ·
* School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Email: cristian.zamfir@epfl.ch
† University of California, Berkeley
Email: {galtekar, istoica}@eecs.berkeley.edu

***Abstract*—Debugging data-intensive distributed applications running in datacenters is complex and time-consuming because developers do not have practical ways of deterministically replaying failed executions. The reason why building such tools is hard is that non-determinism that may be tolerable on a single node is exacerbated in large clusters of interacting nodes, and datacenter applications produce terabytes of intermediate data exchanged by nodes, thus making full input recording infeasible.**

We present ADDA, a replay-debugging system for datacenters that has lower recording and storage overhead than existing systems. ADDA is based on two techniques: First, ADDA provides *control plane determinism*, leveraging our observation that many typical datacenter applications consist of a separate "control plane" and "data plane", and most bugs reside in the former. Second, ADDA does not record "data plane" inputs, instead it synthesizes them during replay, starting from the application's external inputs, which are typically persisted in append-only storage for reasons unrelated to debugging.

We evaluate ADDA and show that it deterministically replays real-world failures in Hypertable and Memcached.

***Keywords*—debugging; record-replay; reliability; datacenter; storage;**

I. Introduction

More and more applications that we use on a daily basis, such as Web search, e-mail, social networks, 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, and Memcached that run on commodity hardware clusters consisting of as many as thousands of machines. As users are growing more dependent on these hosted services, the frameworks and applications employed by the services 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 many nodes. Moreover, such applications typically operate on many terabytes of data daily and are required to maintain high throughput, which makes it hard to record for potential later debugging what they do.

A cluster-wide replay-based solution is the natural option for debugging, as it offers developers the global view of the application: by deterministically replaying a previously-encountered failure, one can use a debugger to zoom in on various parts of the system and understand why the failure occurred. If a cluster-wide replay is not possible, the developer has 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 does not exist in clusters of commodity hardware) or complex algorithms to capture consistent snapshots ([1]).

Developing an automated record-replay debugger is harder for datacenter applications than for a single node, due to the inherent recording overheads. First, these applications are typically *data-intensive*, and the volume of data they process increases proportionally with the size of the system, the power of individual nodes (e.g., more CPUs means more data flowing through), and ultimately with the success of the business. Recording to persistent storage 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 execution. Knowing a priori which dependencies matter is undecidable. A third challenge is that, at large scale, the runtime overhead of a record-replay system has important financial consequences: making up for a 50% throughput drop requires doubling the size of the datacenter. When operating large datacenters it becomes cheaper to hire more engineers to do manual debugging than to increase the size of the datacenter to tolerate the overhead of an automated record-replay system.

Existing work in distributed system debugging does not offer solutions to these challenges. Systems like Friday [2] address distributed replay, but have high overhead for data-intensive datacenter applications.

To address these challenges, we developed ADDA, an automated replay-based debugging system for dat-

978-1-4799-0181-4/13/$31.00 ©2013 IEEE

acenter applications. Three ideas make ADDA more efficient than existing systems. First, a large class of datacenter applications are split into a control plane and a data plane, and most bugs reside in the former [3], so focusing on deterministically replaying the control plane enables the debugging of most problems. Second, there is a class of datacenter applications for which external inputs are anyway persisted in append-only storage (e.g., for compliance, fault tolerance) and thus are available when debugging. When combined with the ability to replay the control plane, this property allows ADDA to do record-replay by recording just a small subset of all inputs. Third, with suitable recording, it is possible to deterministically synthesize (regenerate) all intermediate data sets during debugging, thus eliminating the need to record intermediate data.

In this paper, we make three contributions:

- A technique for *recording the behavior* of datacenter applications with lower overhead than existing systems;
- A technique for *synthesizing intermediate data* to enable replay-based debugging, in a way that is not affected by nondeterminism;
- A technique called *reduced-scale replay*, which allows replaying a failed execution that occurred in the production cluster on a smaller cluster or on a subset of the original cluster.

In the rest of the paper, we give an overview (§II), present ADDA's design (§III) and prototype (§IV), we evaluate ADDA (§V), discuss related work (§VI), and end with a discussion (§VII) and conclusions (§VIII).

II. Overview

Many replay debugging systems have been built over the years, and experience indicates that they are invaluable in reasoning about nondeterministic failures [2], [4], [5], [6], [7], [8], [9], [10], [11]. However, we believe no existing system meets the demands of the datacenter environment. We discuss these requirements next.

A. Design Requirements

a) Whole-System Replay: The system should be able to replay the behavior of *all nodes* in the distributed system, if desired. 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 occurred at the system level.

b) Low Recording Overhead: Large datacenters consist of hundreds or thousands of machines. In such large systems, even a moderate recording overhead can translate into significant operation and capital costs. Thus, low recording overhead should be a major goal when replay-debugging datacenter applications.

c) 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.

d) Minimal Setup Assumptions: A replay-debugging system should record and replay user-level applications with no administrator or developer effort. It should not require special hardware, languages, or source-code analysis, and no modifications to the applications themselves. Datacenters may have components that must be treated as black boxes (e.g., if source code is not available), but still need to be replayable. Special languages and source-code modifications (e.g., custom APIs and annotations, as used in R2 [12]) are cumbersome to learn, maintain, and retrofit onto existing datacenter applications. Source-code analysis is often not possible, since some components may be closed-source. Finally, datacenter applications operate in a "mixed world": while the nodes running the application can be assumed to be recorded, other nodes (e.g., those running DNS servers) may not be.

e) Debug Determinism: To be useful, a replay-debugging system must reproduce a production failure and its root cause [13] for most failures, 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 [3]).

B. Ideas Enabling Our Solution

1) Control-Plane Determinism Suffices: The key idea 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 same *control plane* behavior as the original.

The control plane of a datacenter application is the code that manages data flow and implements operations like locating a particular block in a distributed filesystem, maintains replica consistency in a meta-data server, or updates the routing table of a software router. Control plane operations tend to be complicated: they account for over 99% of the bugs in datacenter applications [3]. On the other hand, the control plane accounts for less than 1% of all datacenter network traffic [3].

In contrast, datacenter application debugging rarely requires reproducing the same *data plane* behavior [3]. The data plane is the code that processes the data (e.g., that computes checksums of an HDFS filesystem block or searches for a string as part of a MapReduce job). In contrast with the control plane, data plane operations are simple: they account for under 1% of the code in a datacenter application [3] and are often part of well-tested libraries. Yet, the data plane generates and processes over 99% of datacenter traffic [3]. Thus, unless the root

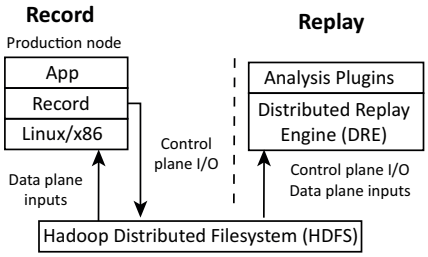

Fig. 1: ADDA's architecture. It uses the recorded control plane inputs and the persistently stored data plane inputs to generate, in a best effort fashion, a control-plane deterministic run.

cause is in the data plane code, reproducing the same data plane behavior is not necessary.

We observed that the separation between control plane and data plane holds for a representative class of datacenter applications, such as CloudStore [14], MapReduce/Hadoop [15], Memcached [16], Cassandra [17], and Hypertable [18]. However, for many other datacenter applications, the 1%-99% separation may not hold, therefore ADDA does not achieve low runtime overhead for these applications.

Our hypothesis is that by deterministically replaying the control plane, ADDA can reproduce most bugs with low-overhead recording. We verified this hypothesis for the bugs in our evaluation.

2) Data Plane Inputs Are Persistently Stored: The data that enters the system from outside is often stored persistently in append-only file systems, such as HDFS. Thus, we can assume that these external inputs are available during debugging. Thus, ADDA does not need to record these external inputs. This is a crucial property of a large number of popular datacenter applications, as it obviates ADDA from recording prohibitive amounts of data. ADDA mainly targets such applications.

III. Design

Having described the main insights behind our approach, we now describe ADDA's design.

A. 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 replayed runs will exhibit identical control plane behavior to that of the original run. Control plane determinism makes datacenter replay practical because it circumvents the need to record data plane communications (which have high data-rates), thereby allowing ADDA to efficiently record the execution on all nodes in the system.

ADDA's architecture is given in Fig. 1. It operates in two phases: record mode and replay mode.

1) Record Mode:

a) What ADDA *records:* All ADDA-enabled nodes record control plane nondeterministic events, 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 control plane, thus they are also recorded. Control plane nondeterminism 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).

b) 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 I/O is stored persistently and is available during replay. ADDA does not assume that internal data plane I/O is stored persistently. Instead, it attempts to regenerate it during replay.

c) The recording log: ADDA stores the recording in a local log file on each node and asynchronously transfers the logs to a distributed file system such as HDFS (see Fig. 1). If the datacenter application supports consistent snapshots, the logs can be truncated, such that replay can start from the latest snapshot instead of the beginning of the recording.

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: DRE guarantees replay of control plane nodes (the most complex and bug-prone components), but 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 using the following principles:

a) Synthesize missing non-deterministic inputs when possible: While recording control plane nondeterminism is sufficient for replaying control plane nodes in a distributed application, mixed control/data plane nodes (e.g., a Hypertable range server) require recording data plane nondeterminism to deterministically replay even their control plane components. The DRE attempts to recompute this unrecorded nondeterminism in a best-effort fashion using Data Plane Synthesis (§III-C).

b) Provide a platform for automated debugging: Going beyond replay, the DRE also serves as a platform for writing powerful replay-mode analysis plugins for sophisticated distributed analyses, such as distributed data flow and global invariant checking (§III-D).

B. Recording Control Plane Non-determinism

To record control plane non-determinism, ADDA must first identify it. In the general case of an arbitrary application, 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. Thus, ADDA semi-automatically classifies control plane nondeterminism and then interposes on communication channels (§III-B1) to record the ordering and the values of the inputs only for channels classified as control plane (§III-B2).

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 can be easily interposed efficiently, as they operate through well-defined interfaces (system calls). Interposition is 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. Other sources of non-determinism (e.g., local time and random number generators) are recorded similarly.

Shared memory channels are harder to interpose efficiently because this requires detecting sharing (i.e., 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 accesses. This is expensive, since it requires intercepting all memory accesses.

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 [19] and later implemented and refined by SMP-ReVirt [7]. Page-based CREW leverages page-protection hardware found in modern MMUs to detect concurrent accesses to shared pages. When CREW detects concurrent accesses on a shared page, it serializes the accesses to the page. Details of our CREW implementation are given in §IV-B1.

If the application does not have internal non-determinism caused by data races, ADDA can just record the order of synchronization operations, with substantially lower overhead. However, the applications we evaluated (§V) do have data races.

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 [3]. For example, Hadoop [15] 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).

It may not be practical for the developer to annotate all control plane channels. 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 infer 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.

The token-bucket classifiers on socket, pipe, tty, and file channels are parameterized with a token fill rate of 100KBps and a maximum size of 1MB.

Shared-memory channels: The data-rates here are measured in terms of CREW-fault rate (the rate at which CREW serializes accesses to shared pages). 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/second and a burst of 1000 faults/second were enough to identify control plane sharing (§V).

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 on shared-memory channels (e.g., spin-locks) may not be captured, which may preclude correct replay. 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: on a lock miss, `pthread_mutex_lock()` will await notification of lock availability in the kernel instead of spinning.

3) Taming False Sharing with Best-Effort CREW: Under certain workloads, the CREW protocol can incur high page-fault rates that will seriously degrade performance. 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 this case, CPUs do not actually communicate on a channel.

Regardless of the cause, ADDA uses 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 several 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 freedom assumption is met for those pages (ADDA records the lock order to handle this case).

C. 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 despite not having recorded the original data plane inputs.

To address this challenge, the DRE employs a novel technique we call Data Plane Synthesis (DPS). DPS works under the assumption that external data plane inputs are persistently stored in append-only storage by the application and thus available later during replay (for example, click logs that get saved for further analytics). DPS regenerates the communication on data plane channels using the stored data plane inputs. DPS enables ADDA to synthesize data plane inputs during replay without recording them in production.

1) Regenerating Intermediate Inputs: The external data plane inputs can be used to replay those processes that read them directly (i.e., front-end systems). Typical front-end systems transform these external inputs and pass them to internal/intermediate nodes. ADDA does not record these intermediate inputs, instead it regenerates them using the classic technique of order-based replay [19]: 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 is to apply order-based replay to all internal/recorded nodes. We describe inductively how ADDA provides order-based replay:

Base Case. Replay the data plane outputs of a single node, given access to all inputs. As long as shared memory non-determinism on the node is replayed, the node will generate the same data plane outputs as the original execution.

Inductive Step. Given n order-replayed nodes $A_1, A_2, ..., A_n$, whose outputs are inputs to node B, ADDA ensures that B gets the merged outputs of $A_1, A_2, ..., A_n$ in the original order. Since shared memory non-determinism on B is replayed, B generates the original outputs because they are fully determined by $A_1, A_2, ..., A_n$ and the execution of B.

2) Dealing with a Mixed World: In an ideal world, all nodes in the datacenter would be using ADDA. In reality, only some of the nodes (the ones running a particular datacenter application) 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 thus may behave differently at replay time.

ADDA handles these two aspects separately:

a) Persistent-Store Nondeterminism: Since the persistent store housing data plane inputs are not traced, DPS faces the following challenge:

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. For instance, HDFS uses its own interface, which is not compatible with VFS, and may redistribute blocks or even alter block IDs. Hence, a naive approach that simply reissues HDFS requests with original block IDs will produce nondeterministic results.

ADDA addresses this challenge 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 using the socket-based HDFS protocol directly. The VFS layer addresses the 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., HDFS block assignments and IDs).

b) Network Nondeterminism: ADDA does not record and reproduce low-level 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 run, making it hard for DPS to determine where to send messages to. For example, ADDA's reduced-scale replay enables replaying a 1000 node cluster on fewer (e.g., 100) nodes, and some of these replay nodes will have different IP addresses. The second challenge is that the network may unpredictably drop messages (e.g., for UDP datagrams). This means that simply re-sending 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.

c) REPLAYNET*:* As with persistent-store non-determinism, ADDA shields DPS from network non-determinism using a layer of indirection. ADDA introduces REPLAYNET, a virtual replay-mode network that

abstracts away the details of IP addressing and unreliable delivery: rather than sending messages directly through the physical network at replay time, ADDA sends messages through REPLAYNET. At the high level, REPLAYNET can be thought of as a key-value store that maps unique message IDs to message contents. To send a message over REPLAYNET, a node simply inserts the message contents into the key-value store 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.

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 can record the message ID since the sender piggy-backs it on the outgoing message at record time. Further details of piggy-backing are given in §IV-B3.

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 at replay time, a node first consults 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) Coping with Unrecorded Shared-Memory Ordering: ADDA ensures that the components that are only part of the control plane (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 the inputs of the 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. The ability to replay control plane nodes is still valuable because the control plane accounts for most bugs [3].

A key requirement of order-based replay is that complete ordering information must be available. Unfortunately, ADDA's recording of shared memory interleavings may be incomplete, since ADDA disables CREW for high data-rate pages (§III-B3). In particular, the interleaving of data races on such pages is not recorded, hence precluding the reproduction 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.

D. Enabling Automated Debugging

In addition to replay, ADDA provides a powerful platform for building powerful replay-mode, automated debugging tools. ADDA was designed to be extended via plugins, hence enabling developers to write sophisticated distributed analyses that would be too expensive to run in production. We created several plugins using this architecture, including distributed data flow analysis, global invariant checking, communication graph analysis, and distributed-system visualization.

We describe ADDA's plugin model, and then describe a simple automated-debugging plugin for distributed data flow analysis.

1) Plugin Model: A key goal of ADDA's plugin model is to ease the development of sophisticated plugins. Therefore, ADDA plugins are written in Python and provide the following properties.

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 (IDs are generated during the recording):

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

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 non-deterministic results.

Access to fine-grained analysis primitives. ADDA is pre-loaded with commonly-used, fine-grained analysis primitives. An example of such a primitive is ADDA's data-flow analysis primitive, which exports two functions (`is_tainted(node, addr)` and `set_taint(node, addr)`) that plugins can invoke to determine if the byte of memory at address `addr` is tainted by an external data source and to set the byte of memory at address `addr` as tainted.

2) Distributed Data Flow Plugin: DDFLOW is a distributed data flow analysis plugin for ADDA. DDFLOW provides 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 diagnosing bugs that lead to data loss: it allows developers to track the flow of data and helps quickly identify where data ends up – a process that could take hours if done manually. DDFLOW highlights the power of ADDA plugins because it is an analysis that is too heavyweight to do in production, but can easily be done during replay. The DDFLOW plugin can be written in just a few lines (initialization code is omitted):

```
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]
```

IV. Implementation

We implemented ADDA for clusters of Linux x86 machines. ADDA consists of approximately 150 KLOC of source code (40% LibVEX [20] and 60% ADDA + plugins). We show how to use ADDA and then discuss the main implementation challenges we encountered.

A. Usage

The first step in using ADDA consists of recording; to start this phase, a user invokes ADDA on the application binary, specifying the location of the log (e.g., distributed storage) and the location of persistent data files (e.g., an HDFS mount):

```
$ adda-record --save-as=hdfs://host/demo
--persistent-store=/mnt/hdfs/data ./application
```

ADDA will then record the application, without recording data plane inputs originating from the specified persistent storage.

To replay using the DDFLOW analysis plugin (§III-D2), one only specifies the plugin name and the location of previously collected recordings:

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

B. Lightweight Recording of User-Level Code

Bugs in datacenter applications often reside in application code rather than kernel code. ADDA therefore only records the non-determinism needed to replay user-level code of developer-selected application processes.

1) Interpositioning: ADDA interposes on user-level communication channels (sockets, pipes, files, and shared-memory) of traced processes.

Sockets, pipes, and files are interposed with a kernel module. The module delivers to ADDA a signal for every system call invoked by a traced process. To address the high syscall rates of some datacenter applications, we also intercept syscalls made through Linux's vsyscall page, 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. Conceptually, ADDA's CREW implementation follows that of SMP-ReVirt [7]: it maintains shadow page tables whose permission are upgraded and downgraded at CREW events. 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 CREW overhead due to false sharing in the kernel (a significant bottleneck in SMP-ReVirt). ADDA interposes on page table operations using Linux's paravirt_ops, similarly to Xen.

2) Asynchronous Events: The only asynchronous events ADDA must record are signals and thread preemptions. This is a key benefit of recording at the user level, unlike VM-level replay tools that also have to record all interrupts.

ADDA ensures accurate delivery of asynchronous events during replay (i.e., at the same point in program execution as during recording). One way to do this is to count the number of instructions at record time and deliver the event at the recorded instruction count during replay. This requires the use of a software instruction counter that would incur high runtime overhead. Instead, ADDA precisely identifies a point in program execution via the triple <eip, ecx, branch count>, which can be efficiently obtained from x86 CPUs.

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.

Instead, ADDA uses two techniques, both leveraging the semantics of system calls, to reduce piggy-backing overheads: *message-level piggy-backing* and *TCP-aware unique IDs*. Message-level piggy-backing leverages the observation that data plane channels send data in large chunks (e.g., Memcached performs sys_send on 2 MB buffers), so ADDA piggy-backs at the message level instead of packet level. ADDA leverages the observation that datacenter applications typically use TCP to transfer data, and each message in a TCP stream has a unique ID within the stream (i.e., its 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.

C. Distributed Replay and Analysis

1) Serial Replay: The current ADDA prototype provides the illusion of serial replay by 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 overhead will increase linearly with the number of 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 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.

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.

A key challenge in replaying in binary translated mode is that LibVEX does not simulate operations on hardware performance counters, which are used by ADDA 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).

V. Evaluation

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

A. Experience

In this section we describe how we used ADDA to succesfully reproduce and debug bugs in Hypertable [21]. Hypertable [21] is an open source, high performance data store designed for large-scale data-intensive tasks and is modeled after Google's Bigtable [22]. Hypertable is deployed at Baidu, the leading search services in China, and the Rediff online news provider.

1) Hypertable Hang Under Memory Pressure: We found 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. 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 hung the master. 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 the particular cluster machine was accidentally configured without a swap partition, making memory allocations more likely to fail.

2) Data Loss in Hypertable: We used ADDA to debug a previously solved Hypertable defect [23] that causes updates to a database table to be lost when multiple Hypertable clients concurrently load rows into the same table. According to Hypertable's bug tracker, this bug took 6 days to fix. The bug is hard to reproduce, and its root cause spans 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 that are not responsible for hosting them (the slave nodes are called Hypertable range servers and they are responsible for holding a piece of the entire data). 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 of 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 40%. Afterwards, we replayed the failure with ADDA in a single-machine setup. We inserted breakpoints during row range migration, where we suspected the root cause to be 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.

B. Recording Efficiency

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

The size of the cluster may not be representative of the size of current datacenters, 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 (§IV-B2). This limitation may be removed by on-going work on virtualizing performance counters [24].

We measure ADDA's recording overhead versus the overhead of the naive approach that records all inputs, in order to show the benefits of DPS. To simulate the naive approach, we configured ADDA to log all inputs. We first evaluate the single processor case (§V-B1), then the logging overhead (§V-B2) and then the multiple

CPU case (§V-B3). We did not compare against mature record-replay solutions, such as VMWare Workstation, since it does not work for multiple CPUs and has been deprecated since version 7.

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

We evaluate on two systems: Memcached and Hypertable.

Memcached [16] is a high-performance, distributed memory object caching system, typically used for speeding up dynamic web applications by alleviating database load. Memcached is used by online services providers such as Youtube, Wikipedia, and Flickr.

To evaluate the efficiency of recording a Memcached deployment, we simulated a photography blog Web application in which Memcached is used by user-facing Web application servers to cache the files containing the photos. This setup resembles the Facebook photo storage [25], in which Memcached is used to reduce latency. We assume that the photos are stored in persistent storage (i.e., HDFS) and the clients (i.e., the user-facing Web applications, which are also running in the same datacenter) 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 runs on a separate machine. Each client randomly selects one of the Memcached servers to either read or write a photo—reads are selected with 90% probability and writes with 10% probability, since reads are predominant in Facebook's daily photo traffic [25].

For this experiment we used a setup consisting of 4 Memcached servers and 7 clients, each client having 4 threads. Overhead 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.

ADDA's recording overhead with varying size of the input from persistent storage (Fig. 2) 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 DPS: logging all inputs causes the naive approach to have up to 5 times higher runtime overhead than ADDA.

Fig. 3 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 overhead is measured in terms of reduction in client throughput relative to the native execution.

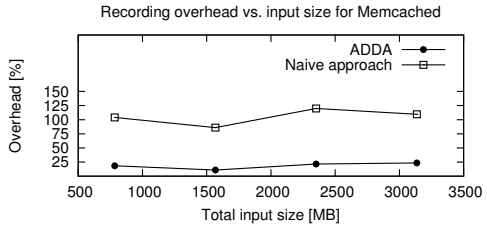

Fig. 2: Recording overhead compared to the native execution in Memcached while varying the total size of the input from persistent storage.

This experiment shows that ADDA's overhead is between 20% and 65%, and scales well with the number of nodes in the system (Fig. 3). 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 have to record all their inputs, the clients become slower, so the impact of heavy recording for the naive approach decreases.

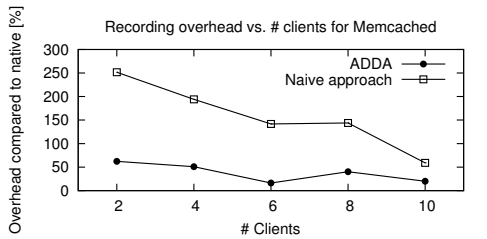

Fig. 3: Recording overhead for Memcached with varying number of clients.

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 generate a workload that would be performed by a user-facing component of the datacenter, such as a Web application server.

The range servers store the content of the database tables in memory and also dump this content to a distributed file system such as HDFS. Because ADDA currently requires that the target application use a VFS-like interface to communicate with the file system (§III-C2) 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.

Fig. 4 shows that, for Hypertable, the recording overhead scales well with the size of the input from persistent storage. The overhead, measured as reduction in transaction throughput, is 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.

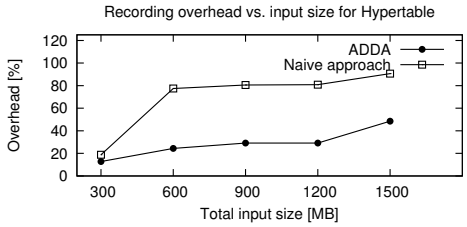

Fig. 4: Recording overhead for Hypertable with varying size of the input from persistent storage.

Fig. 5 shows that ADDA scales well with the number of traced 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.

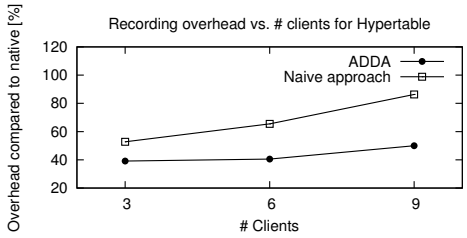

Fig. 5: Recording overhead in Hypertable with varying number of clients.

2) Log Size: ADDA has low logging rates. Fig. 6 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 (inputs exchanged between recorded nodes), 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. Memcached is designed so that server instances do not communicate with each other, otherwise the log size for the naive approach would increase even more, while ADDA does not record this communication.

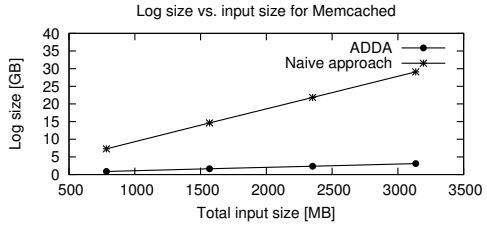

Fig. 6: Log size for recording a Memcached workload with varying input size from persistent storage. ADDA has 10× smaller logs compared to the naive approach.

Hypertable exhibits a similar behavior (Fig. 7). We expect these results to improve even more with a simple optimization: our current DPS prototype allocates a static 15KB entry (or a multiple of this size, if needed) for recording the meta-data associated with an I/O system call. However, a single 15KB entry is typically too large: for Hypertable, log entries are dominated by zeros, which we could compress to 100× smaller size. By adding support for variable entry sizes, we expect ADDA's logging rates to improve substantially.

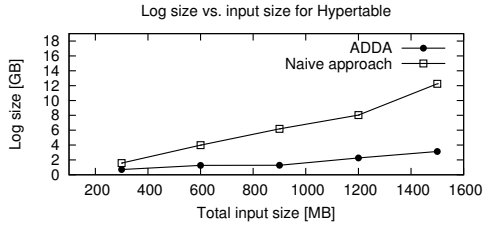

Fig. 7: Log size for recording a Hypertable workload while varying the input size from persistent storage.

3) Performance for Multi-Processors: To validate that our assumption about applying CREW selectively to control plane components (i.e., the Hypertable master and lock server) holds (§III-B), we enabled CREW in the experiment in Fig. 5. Thus, the control plane components were allowed to take advantage of both CPUs of their machines. ADDA had the same overhead 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 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 design choice to turn 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 datacenter applications and we are in the process of evaluating this further.

C. Replay

Replay is serial, therefore replay overhead 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× slower than the original recorded run. We also replayed a Hypertable workload similar to the previous experiments. The Hypertable setup consisted of a lock server, a master server, 2 range servers, and 3 clients. The replay was 2.7× slower than the original run. In both experiments, all inputs were recorded due to a bug that prevented us from doing the replay with `REPLAYNET`. Only these two experiments were done without `REPLAYNET`. However, we observed that, typically for these applications, replay overhead using `REPLAYNET` is similar to the replay overhead of the naive approach. In both these experiments the replay was not $n\times$ slower (where n is the total number of nodes), because replay can fast-forward the execution of some instructions 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.

VI. Related Work

Classic single node replay systems such as Instant Replay [19], VMWare [6], and SMP-ReVirt [7] 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 not only incur throughput losses but also call for additional storage infrastructure (e.g., an additional large scale distributed file system). Moreover, systems like WiDS [4] and Friday [2] provide distributed replay but have high overhead for data-intensive applications.

Several relaxed-deterministic replay systems (e.g., PRES [8] and ReSpec [9]) and hardware and/or compiler-assisted systems (e.g., Capo [11], CoreDet [26]) support efficient recording of multi-core programs. But, like classic systems, they still incur high record rates on network and disk-intensive distributed systems such as datacenter systems.

R2 [12] provides an API and annotation mechanism with which developers may select the application code to be recorded and replayed. R2 could be used to record just control plane inputs, thus incurring low recording overheads. However, the annotations require considerable developer effort to manually identify the components that need to be recorded. On the other hand, `ADDA` makes this selection automatically based on the data-rate heuristic.

MPIWiz [27] is a hybrid deterministic replay system that exploits the traffic patterns of MPI applications to reduce recording overhead by identifying subgroups of MPI processes for which it is more efficient to record all communication messages vs. the order of the exchanged messages. First, MPIWiz addresses MPI applications, while `ADDA` targets datacenter applications. Second, MPIWiz does not handle non-determinism caused by data races and assumes shared memory non-determinism is due only to the order of MPI calls. `ADDA` is more general and captures all sources of non-determinism, including data races. Third, MPIWiz decides to record all communication data at the granularity of a process group, while `ADDA` does it per network channel, using the control/data plane separation.

Replay-debugging systems such as SherLog [28] and ESD [10] can efficiently replay single-node applications while recording very little information, or no information at all. These systems use inference to recompute missing runtime information. However, they were not designed for recording distributed systems, much less so 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 datacenter.

VII. Discussion

For the systems that do not meet our assumptions, the runtime overhead may be deemed to be too high for production use. For such cases, `ADDA` is still useful during development. Our evaluation shows runtime overhead ranging between 10% to 65% (§V). We are not aware of any other record-replay system that has lower overhead for data-intensive datacenter applications. We see opportunities for several simple engineering optimizations and hope that a production-grade implementation of `ADDA` can reduce the overhead to under 10% for an important subset of the datacenter applications and workloads.

Not all applications may meet our two main assumptions about the control/data plane separation and the persistence of external inputs (§II). For instance the running time of a parallel scientific application may be dominated by the control plane code. Moreover, a datacenter application may not store external inputs (such as data acquired by a telescope) to append-only storage simply because of the sheer size of the inputs. On the one hand, for these applications, `ADDA`'s overhead may be unacceptably high. On the other hand, several applications in addition to the applications we evaluated (§V), such as, Hadoop [15], Cassandra [17], CloudStore [14], and applications that process click

streams (in which the initial inputs are logged for audit purposes) do meet ADDA's assumptions. Moreover, the external inputs to MapReduce jobs are typically stored in HDFS, which is append-only storage. This represents an important subset of datacenter applications.

Reduced-scale replay (§III-C) is useful in several common cases. For instance, there are cases in which the original machines are not available for replay anymore (e.g., machines may be down due to hardware failures, or the cluster may be loaded with another job). We designed reduced-scale replay for these cases.

If ADDA does to record an un-synthesizable source of non-determinism (e.g., a data race in the data-plane), the replay might diverge from the recording. ADDA detects divergences by checking that the delivery point of asynchronous events (i.e., the triple <eip, ecx, branch_count>) is the same during record and replay.

ADDA supports recording of multiple multi-processor nodes, but does not yet support multi-processor replay. We plan to add this in future work. Replaying CREW events is done in other systems [7] and is not challenging if all CREW events are recorded. However, if ADDA missed the recording of CREW events (e.g., due to misclassifying control plane code as data plane code), this could lead to divergence during replay, therefore replay may not be possible. In this case, ADDA could use inference-based techniques [8], at the expense of longer replay time.

VIII. Conclusion

We presented ADDA, a replay-debugging system for data-intensive datacenter applications. To reduce recording overhead, ADDA leverages two important observations: that the "control plane" and "data plane" of datacenter applications can be identified and recorded with different determinism guarantees and that external inputs are typically persisted in append-only storage for an important class of datacenter applications. ADDA has low runtime overhead and logging rates, and deterministically replays real-world failures in popular datacenter applications. To achieve this, ADDA records with high accuracy the control plane of a datacenter application and does not record intermediate data that it can synthesize during replay, starting from the application's external inputs. Moreover ADDA can perform reduced-scale replay and run sophisticated analyses during replay.

References

[1] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, 1985.

[2] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," in *Symp. on Networked Systems Design and Implementation*, 2007.

[3] G. Altekar and I. Stoica, "Focus replay debugging effort on the control plane," in *Workshop on Hot Topics in Dependable Systems*, 2010.

[4] X. Liu, W. Lin, A. Pan, and Z. Zhang, "Wids checker: Combating bugs in distributed systems," in *Symp. on Networked Systems Design and Implementation*, 2007.

[5] 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 *Intl. Conf. on Virtual Execution Environments*, 2006.

[6] "VMware vSphere 4 fault tolerance: Architecture and performance," http://www.vmware.com/resources/techresources/10058.

[7] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Intl. Conf. on Virtual Execution Environments*, 2008.

[8] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou, "PRES: Probabilistic replay with execution sketching on multiprocessors," in *Symp. on Operating Systems Principles*, 2009.

[9] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Online multiprocessor replay via speculation and external determinism," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[10] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated debugging," in *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.

[11] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: a software-hardware interface for practical deterministic multiprocessor replay," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[12] 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 *Symp. on Operating Systems Design and Implementation*, 2008.

[13] C. Zamfir, G. Altekar, G. Candea, and I. Stoica, "Debug determinism: the sweet spot for replay-based debugging," in *Workshop on Hot Topics in Operating Systems*, 2011.

[14] "Cloudstore," http://kosmosfs.sourceforge.net.

[15] "Hadoop," http://hadoop.apache.org/.

[16] "Memcached," http://www.memcached.org/.

[17] "Cassandra," http://cassandra.apache.org.

[18] "Hypertable," http://www.hypertable.org.

[19] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 471–482, 1987.

[20] "LibVEX," http://valgrind.org/.

[21] "Hypertable," http://www.hypertable.org/.

[22] 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," *USENIX Annual Technical Conf.*, 2006.

[23] "Hypertable issue 63," http://code.google.com/p/hypertable/issues/.

[24] R. Nikolaev and G. Back, "Perfctr-xen: a framework for performance counter virtualization," in *Intl. Conf. on Virtual Execution Environments*, 2011.

[25] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Symp. on Operating Systems Design and Implementation*, 2010.

[26] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "Coredet: A compiler and runtime system for deterministic multithreaded execution," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[27] Xue, "MPIWiz: Subgroup reproducible replay of MPI applications," in *Symp. on Principles and Practice of Parallel Programming*, 2009.

[28] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.