Focus Replay Debugging Effort on the Control Plane

Gautam Altekar
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

Replay debugging systems enable the reproduction and debugging of non-deterministic failures in production application runs. However, no existing replay system is suitable for datacenter applications like Cassandra, Hadoop, and Hypertable. On these large scale, distributed, and data intensive programs, existing replay methods either incur excessive production recording overheads or are unable to provide high fidelity replay.

In this position paper, we hypothesize and empirically verify that *control plane determinism* is the key to record-efficient and high-fidelity replay of datacenter applications. The key idea behind control plane determinism is that debugging does not always require a precise replica of the original application run. Instead, it often suffices to produce some run that exhibits the original behavior of the *control-plane*–the application code responsible for controlling and managing data flow through a datacenter system.

1 Introduction

The past decade has seen the rise of large scale, distributed, data-intensive applications such as HDFS/GFS [16], HBase/Bigtable [10], and Hadoop/MapReduce [11]. These applications run on thousands of nodes, spread across multiple datacenters, and process terabytes of data per day. Companies like Facebook, Google, and Yahoo! already use these systems to process their massive data-sets. But an ever-growing user population and the ensuing need for new and more scalable services means that novel applications will continue to be built.

Unfortunately, debugging is hard, and we believe that this difficulty has impeded the development of existing and new large scale distributed applications. A key obstacle is non-deterministic failures–hard-to-reproduce program misbehaviors that are immune to traditional cyclic-debugging techniques. These failures often manifest only in production runs and may take weeks to fully diagnose, hence draining the resources that could otherwise be devoted to developing novel features and services [23]. Thus *effective tools for debugging non-deterministic failures in production datacenter systems are sorely needed.*

Replay-debugging technology (a.k.a, deterministic replay) is a promising method for debugging non-deterministic failures in production datacenters. Briefly, a replay-debugger works by first capturing data from non-deterministic data sources such as the keyboard and network, and then substituting the captured data into subsequent re-executions of the same program. These replay runs may then be analyzed using conventional tracing tools (e.g., GDB and DTrace [9]) or more sophisticated automated analyses (e.g., race and memory-leak detection, global predicates [14], and causality tracing [13]).

1.1 Requirements

Many replay systems have been built over the years and experience indicates that they are invaluable in reasoning about non-deterministic failures [5, 8, 12, 14, 15, 18, 19, 21, 25]. However, no existing system meets the demands of the datacenter environment.

Low Overhead Recording. A datacenter replay system must be on at all times during production so that arbitrary segments of production runs may be replay-debugged at a later time.

Unfortunately, replay systems such as liblog [15], VMWare [5], PRES [21] and ReSpec [18] require all program inputs from across all nodes to be logged, hence incurring high throughput losses and storage costs on multicore, terabyte-quantity processing.

High Fidelity Replay. A datacenter replay system should also be able to reproduce the execution of *all nodes* in the distributed system, if needed, with precision sufficient to isolate the root cause of the execution failure.

Replay systems such as ODR [6] (our prior work), ESD [25], and SherLog [24] support efficient datacenter recording, but may take exponential time to generate a replay run, sometimes precluding replay altogether, let alone high-fidelity replay. Annotation-based replay systems such as R2 [17] enable the developer to selectively trade recording overhead and replay fidelity, but provide little guidance in making the right trade-off.

1.2 Hypothesis: The Control Plane is Key

The contribution of this work is a hypothesis and its experimental verification.

The Hypothesis. We put forth the hypothesis that *control plane determinism* is sufficient for debugging

datacenter applications. The key observation behind control-plane determinism is that, for debugging, we do *not* need a precise replica of the original production run. Rather, it generally suffices to produce some run that exhibits the original run's *control-plane* behavior.

The control-plane of a datacenter application is the code that manages or controls data-flow. Examples of control-plane operations include 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. The control plane is widely thought to be the most bug-prone component of datacenter systems. But at the same time, it is thought to consume only a tiny fraction of total application I/O.

A corollary hypothesis is that datacenter debugging rarely requires reproducing the same *data-plane* behavior. The data-plane of a datacenter application is the code that processes the data. Examples include code that computes the checksum of an HDFS filesystem block or code that searches for a string as part of a MapReduce job. The data plane is widely thought to be the least bug-prone component of a datacenter system. At the same time, experience indicates that it is responsible for a majority of datacenter traffic.

Supporting Evidence. We support the above hypothesis with experimental evidence. In particular, we show that, for datacenter applications, (1) the control plane rather than the data plane is responsible for 99% of all bugs in a datacenter application and (2) the data plane rather than the control plane is responsible for 99% of all I/O consumed and generated by a datacenter application. Taken together, these results suggest that, by relaxing determinism guarantees to control-plane determinism, a replay system will be able to provide both low-overhead recording and high fidelity replay.

While our goal is to advocate control plane determinism, we do not discuss the mechanism for achieving it in a real replay system. We address these details in the DCR datacenter replay system [7].

2 Testing the Hypothesis

We present the criteria for verifying our hypothesis and then describe the central challenge in its verification.

2.1 Criteria and Implications

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

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

Data Rates. Second, we must show that the control plane rather than the data plane is *by far* the least data intensive component of datacenter systems. If so, then a control plane deterministic replay system is likely to incur negligible record mode overheads – after all, such a system need not record data plane traffic [7]. If, however, the control plane has high data rates, then it is likely to be too expensive for the datacenter, and our hypothesis will be falsified.

2.2 The Challenge: Classification

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

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

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

Details of our distributed taint-flow analysis are given in Section 2.2.1, while potential flaws with the method are discussed in Sections 2.2.2 and 2.2.3.

2.2.1 Tracking User Data Flow

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

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

Propagating Taint. Unlike single-process taint-flow analyses such as TaintCheck [20], our analysis must track taint both within a node (e.g., through loads and stores) and across nodes (e.g, through network messages).

Within a Node. We propagate taint at byte granularity largely in accordance with the taint-flow rules used by other single-node taint-flow analyses [22]. For instance, we taint the destination of an n-ary operation if and only if at least one operand is tainted. Our analysis does, however, differ from others in two key details. First, we create new taint only when bytes are read from designated user data files (as opposed to all input files or network inputs). And second, we do *not* taint the targets of tainted-pointer dereferences unless the source itself is tainted (this avoids misclassifying control plane code, see Section 2.2.2).

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

We piggyback meta-data on outgoing UDP and TCP messages with the aid of a transparent message tagging protocol we developed in prior work [15]. For UDP messages, the protocol prefixes each outgoing UDP message with meta-data and removes it upon reception. For TCP messages, the protocol inserts meta-data into the stream at `sys_send()` message boundaries, along with the size of the message. On the receiving end, the protocol uses the previous message's size to locate the meta-data for the next message in the stream.

Reducing Perturbation. A key difficulty in performing taint-analysis on a running system is that the high overhead of analysis instrumentation (approximately 60x in our case) severely alters system execution. For instance, in our experiments with OpenSSH [4], taint-flow instrumentation extended computation time so much that `ssh` consistently timed out before connecting to a remote server. This precluded any analysis of the server.

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

To obtain a replay execution suitable for offline taint-flow analysis, we employ the Friday distributed replay and analysis platform [14]. Friday records a distributed system's execution and replays it back in causal order (i.e., respecting the original ordering of sends and receives). Friday was not designed for datacenter operation – it records both control and data plane inputs and hence is too expensive to deploy in production. Nevertheless, it is sufficient for the purposes of collecting and analyzing production-like runs.

2.2.2 Accuracy

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

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

User Data Misidentification. It is possible that we may fail to identify user data files. As a result, some data plane code will be erroneously classified as control plane code. We may also mistakenly designate non-user data files as user-data files. In that case, control plane code will be misclassified as data plane code. Despite these dangers, we note that the possibility of misidentification-identification is very low in practice: our evaluation workloads are composed of only a few user data files that we hand picked (see Section 3.1).

Tainted Pointers. Our policy of not tainting the targets of tainted-pointer dereferences (unless the source itself is tainted) may result in data plane code being misclassified as control plane code. An example is the following snippet from a C implementation of CRC32 used in OpenSSH [4]:

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

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

the corresponding table entry is a constant, and thus our analysis will assume that `crc` is untainted as well.

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

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

Both pointer sensitive and insensitive policies will correctly classify the hash computation as a data-plane operation. However, a pointer-sensitive policy will also classify the lock acquisition (a control plane operation) as a data plane operation. Reads of the lock variable must be dereferenced by the tainted hash code, after all. Unfortunately, such code is common in some applications we've worked with (e.g., Hypertable [2]).

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

Impact on Results. Overall, the above imprecisions in our method are more likely to induce under-tainting rather than over-tainting. In other words, we are more likely to misclassify data plane code as control plane code. Such misclassification will produce unsound bug rate results. In particular, if we observe a high control plane bug rate, then all of those bugs may not stem from control plane code–some, perhaps a sizeable portion, may stem from data plane code. By contrast, the data rate results will remain sound despite under-tainting. Specifically, if we observe a high data plane rate (as we indeed do, see Section 3.3), then those results are accurate. After all, under-tainting can only decrease the measured data plane rate.

2.2.3 Completeness

The results produced by our classifier do not generalize to arbitrary program executions. The reason is that our taint-flow analysis is dynamic rather than static, and therefore we have no way to classify instructions that do not execute in a given run. Though we cannot completely overcome this limitation, we compensate for it by performing our taint-flow analysis on multiple executions with a varied set of inputs (see Section 3.1) We ultimately classify only those instructions executed in at least one of those runs. In future work, we hope to increase the quantity and quality of inputs to derive a more general result.

3 Evaluation

We evaluate our hypothesis on real datacenter applications per the criteria given in Section 2.1. In short, we found that both clauses of our testing criteria held true. That is, we found that control plane code is the most complex and bug prone (with an average per-execution code coverage and reported bug rate of 99%), and that data plane code is the most data intensive (accounting for an average 99% of all application I/O).

3.1 Setup

Applications. We test our hypothesis on three real-world datacenter applications: *CloudStore* [1], *Hyptertable* [2], and *OpenSSH* [4].

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

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

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

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

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

Figure 1: Plane code complexity as a percentage of the number of static x86 instructions that were executed at least once in our runs. As hypothesized, the control plane accounts for almost all of the code in a datacenter application.

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

Figure 2: Plane code complexity as measured by the number of C/C++ functions hosting the top 90% of the most executed instruction locations in a plane. The control plane is more complex in that draws upon a vast array of distinct functions to carry out its core tasks, while the data plane relies on just a handful.

3.2 Bug Rates

Metrics. We gauge bug rates with two metrics: *plane code size* and *plane bug count*. Plane code size is the number of static instructions in the control or data plane of an application, as identified by our classifier (see Section 2.2). Code size is a good approximation of code bug rate since it indirectly measures the code's complexity and thus its potential for defects. Plane bug count is the number of bug reports encountered in each component over the system's development lifetime, and serves as direct evidence of a plane's bug rate.

We measured plane code size by looking at the results of our classification analysis (see Section 2.2) and counting the number of static instructions executed by each plane across all test inputs. We measured the plane bug count by inspecting and understanding, *at the high level*, all reported, non-trivial defects in the application's bug report database. For each defect, we isolated the relevant code and then used our understanding of the report and our code classification to determine if it was a control or data plane issue.

Code Size Results. Figure 1 gives the measured size in static instructions for the control and data planes. At the high level, it shows that almost all of an application's code–99% on average–is in the control plane. Components such as the Hypertable Master and Metadata servers are entirely control plane. This is not surprising because these components don't access any user data; their role, after all, is to direct the placement of user data kept by the Range server. More interestingly, however, components that do deal with user data (e.g., the Hypertable Range server) are still largely control plane.

To understand why the control plane dominates even in the data intensive application components, we counted the number of distinct functions invoked by each plane. The results, shown in Figure 2, reveal that control plane code invokes many functionally distinct operations. For instance, we found that CloudStore's control plane must allocate and deallocate memory (calls for `malloc()` and `free()`), perform lookups on the directory tree (`Key::compare()`) to determine data placement, and prepare outgoing messages (`TcpSocket::Send()`), just to name a few. By contrast, Figure 2 shows that the data plane has extremely low function complexity: one function, in most cases, does almost all of the data plane work. To give an example, we found that almost all of the CloudStore Client's data plane activity consists of calls to `adler32()` – a data checksumming function.

Bug Count Results. Figure 3 gives the number of bug reports for each plane. At the high level, it shows that an average 99% of bug reports stem from control plane errors. We were able to identify two reasons for this result.

The first reason is that significant portions of control plane code is new and written specifically for the unique

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

Figure 3: Plane bug count. The control plane accounts for almost all reported bugs in an application. CloudStore numbers are not given because it does not appear to have a bug report database.

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

Figure 4: The percentage of dynamic x86 instructions issued from well-tested libraries (e.g., code found in `libc`, `libstdc++`, `libz`, `libcrypto`, etc.) and inlined template code (from C++ header files), broken down by control and data planes. The data plane relies almost exclusively on well-tested code, while the control plane contains sizable portions of custom code.

and novel needs of the application. By contrast, the data plane code generally relies *almost exclusively* on previously developed and well-tested code bases (e.g., libraries). To substantiate this, we measured the percentage of instructions executed from within libraries and inlined C++/STL code by each plane. The results, given in Figure 4, show that a median 99.8% of instructions executed by the data plane come from well-tested libraries such as `libc` and `libcrypto`, while only a median 93% of instructions executed by the control plane come from libraries.

A second reason for the high control plane bug count is complexity. That is, the control plane tends to be more complicated. This is evidenced not only by the function complexity results in Figure 2, but also by the nature of the bugs themselves. In particular, our inspection of the source code revealed that control plane bugs tend to be more complex than data plane bugs–an artifact, perhaps, of the need to efficiently control the flow of large amounts of data. For instance, Hypertable migrates portions of the database from range server to range server in order to achieve an even data distribution. But the need to do so introduced Hypertable issue 63 [3] — a data corruption bug that triggers when clients concurrently access a migrating table.

3.3 Data Rates

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

Results. Figure 5 gives the data rates for the control and data planes. At the high level, the results show that the control plane is by far the least data intensive component. Specifically, the control plane code accounts for an average 1% of total application I/O in components that have a mix of control and data plane code (e.g., Hypertable Slave and Client). Moreover, in components that are exclusively control plane (e.g., the Hypertable Master), the overall I/O rate is orders of magnitude smaller than those that have data plane code. These results highlight a key benefit of a control plane deterministic replay system: it provides a drastic reduction in logging overhead that in turn enables low-overhead, always-on recording.

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

Figure 5: Input/output (I/O) traffic size in gigabytes broken down by control and data planes. For application components with high data rates, almost all I/O is generated and consumed by the data plane.

4 Conclusion

A replay debugger for datacenter applications must reproduce distributed system bugs and provide lightweight recording. In this paper, we've argued that a datacenter replay system can do both by shooting for control plane determinism–the idea that is suffices to produce some run that exhibits the original run's control plane behavior. To support our argument, we provided experimental evidence suggesting that the control plane is responsible for most bugs and that it operates at low data rates. Taken together, these results support our position that control plane determinism enables practical datacenter replay.

5 Acknowledgements

We thank the anonymous reviewers and Cristian Zamfir for their feedback. This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Facebook, Hewlett-Packard, Network Appliance, and VMWare, and by matching funds from the State of California's MICRO program (grant 06-149) and the UC Discovery grant COM07-10240.

References

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

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

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

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

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

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

[7] ALTEKAR, G., AND STOICA, I. Dcr: Replay debugging for the data center. Tech. Rep. UCB/EECS-2010-74, EECS Department, University of California, Berkeley, May 2010.

[8] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *VEE* (2006).

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

[10] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).

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

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

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

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

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

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

[17] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008).

[18] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Online multiprocessor replay via speculation and external determinism. In *ASPLOS* (2010).

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

[20] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS* (2005).

[21] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP* (2009).

[22] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Oakland* (2010).

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

[24] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

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