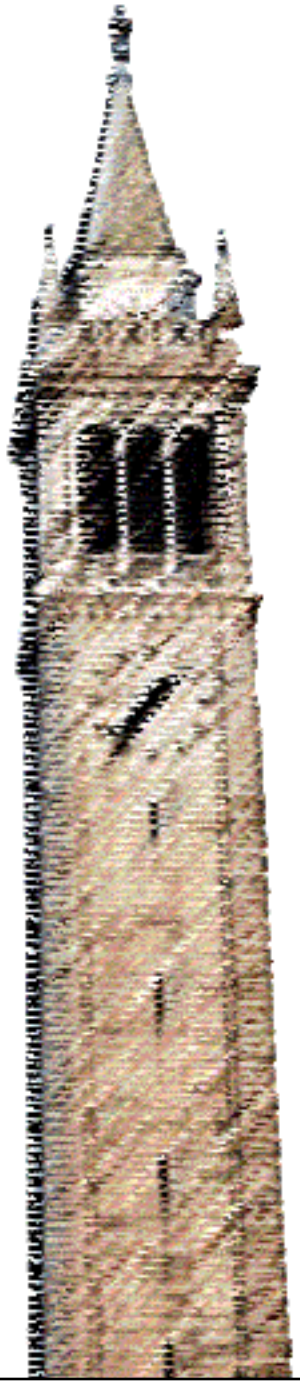
Focus Replay Debugging Effort On the Control Plane



Gautam Altekar
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-88
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-88.html

May 29, 2010

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 [15], HBase/Bigtable [9], and Hadoop/MapReduce [10]. 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 [22]. 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 [8]) or more sophisticated automated analyses (e.g., race and memory-leak detection, global predicates [13, 18], and causality tracing [12]).

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 [4, 7, 11, 13, 14, 17–19, 21, 24]. 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 [14], VMWare [4], PRES [21] and ReSpec [17] 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 program execution on *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 [5] (our prior work), ESD [24], and SherLog [23] support efficient datacenter recording, but may take exponential time to generate a replay run (even for a single node), often precluding replay, let alone high-fidelity replay. Annotation-based replay systems such as R2 [16] enable the developer to selectively trade off recording overhead and replay fidelity, but provide no assurance that the developer-selected tradeoffs will enable isolation of the root cause.

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 error-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 error-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 the determinism guarantees control-plane determinism, a replay system can 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 [6].

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.

Error Rates. First, we must show that the control plane rather than the data plane is *by far* the most error prone component of datacenter systems. If the control plane is the most error prone, then a control-plane deterministic replay system will have high replay fidelity–it will be able to reproduce most application errors. 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 [6]. 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 systems is likely to provide unreliable and irreproducible 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 and annotate user data. By user data we mean any data inputted to the datacenter application with semantics clear to the user but opaque to the system. In the second phase, we automatically identify the static program instructions that are tainted by the annotated user data. Any instructions tainted by data are classified as data plane instructions; the remaining are classified as control plane instructions.

2.2.1 Annotating User Data

To annotate user data we employ two source code annotation functions: `MARK_CHANNEL` and `MARK_MEMORY`.

`MARK_CHANNEL(fd)` takes as input a file descriptor for a file or socket. It's most useful when all data transfered on a particular file or socket channel is known to be user data. For example, suppose we want to denote that all data a Hypertable [2] client reads from a database input file should be marked as user data. Then we may do so as follows:

```
LoadDataSource::LoadDataSource(
      std::string fname, ...) {
   m_fin.open(fname.c_str());
   MARK_CHANNEL(m_fin.get_fd());
```

```
}
```

In the above example, the annotation tells our classification runtime that data returned by all subsequent transactions (i.e., reads and writes) on the recently opened file descriptor should be treated as user data.

`MARK_MEMORY(addr, len)` takes as input a memory region. It's most useful when channel transactions may contain either control or data plane information. For example, if updates received by Hypertable Range server are to the global table directory (METADATA table), then those updates are part of the control plane–they operate on an internal, non-user table. But if updates are to a user-created table, then they are part of the data plane. The `MARK_MEMORY` annotation allows us to make the distinction as follows:

```
void RangeServer::update(
      TableIdentifierT *table,
      BufferT &buffer) {

 const bool is_root =
  strcmp(table->name, "METADATA") == 0;

 if (!is_root) {
   MARK_MEMORY(buffer.buf, buffer.len);
 }
 ...
}
```

In the above code, we mark the incoming data as user data only if the target table is not the root (i.e., METADATA) table.

2.2.2 Tracking User Data Flow

We employ a simple instruction-level, dynamic taint flow analysis [20] to track user data. We chose a dynamic rather than a static approach for two reasons. First, we wanted accurate results. Unfortunately, a static approach is prone to false positives and/or negatives due to pointer aliasing issues. Second, we wanted to track user data through the entire application, including dynamically linked libraries and any dynamically generated code. Many applications employ both for control and data plane processing (e.g., Hadoop), and ignoring them would likely pollute our results. Unfortunately, both techniques have traditionally been difficult for static analysis.

2.2.3 Classification Accuracy

Though we believe our method to be more reliable than manual classification, it has two problems that may foil its accuracy.

First, it is possible that we may fail to annotate some user data entry points. As a result, some data plane code will be erroneously classified as control plane code. This means that if we observe a high control plane error rate, then all those errors may not stem from control plane code–some, perhaps a majority, may stem from data plane code. We note that, in practice, there are only a handful of user data entry points in a datacenter system, and consequently, the possibility of such misclassification is low. We also note that results for control plane data rates are sound even if there is misclassification, because data-plane misclassification can only increase the control plane data rate.

The second limitation is that we perform a dynamic rather than static analysis. This is problematic because, if an instruction isn't executed in one run, then we have no way to classify it. We compensate for this problem by performing the dynamic tainting on multiple executions with a varied set of inputs, ultimately classifying only those instructions executed in at least one of those runs. Of course, it's possible that an instruction's classification may vary from run to run. We workaround this issue by considering the instruction as part of the control plane only if it is consistently untainted across executions.

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 the testing criteria held true. That is, we found that control plane code is the most complex and error prone (with a code coverage of 99% and a 93% bug rate), and that data plane code is the most data intensive (accounting for 99% of all application I/O).

3.1 Setup

Applications. We test our hypothesis on two real-world datacenter applications: *Cloudstore* [1] and *Hyptertable* [2].

Cloudstore is a distributed filesystem written in 40K lines of multithreaded C/C++ code. It consists of 3 sub-programs: the master server, slave server, and the client. The master program 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 4 key sub-programs: the master server, metadata server, slave server, and client. The master and metadata servers coordinate the placement and distribution of database tables. The slaves store and serve the contents of tables placed there by clients.

Workloads and Testbed. We chose the workloads to mimic datacenter operation. 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 2 clients concurrently get and put 10 GB gigabyte files. We used 1 master server and 1 slave server.

3.2 Error Rates

Metrics. We gauge error 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 an accurate approximation of code error 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 error 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 plane bug count by inspecting and understanding all 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.

Results. Figure 1(a) gives the measured size in static instructions for the control and data planes, while Figure 1(b) gives the number of bug reports for each plane. At the high level, these figures convey two key results.

First, 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 because they don't access any user data; their role is to mange the user data kept by the Range server. More interestingly, those components that do deal with user data (e.g., the Hypertable Range server) are still largely control plane. This makes sense as, aside from hashing row keys and compressing tables, most of the Range server code is devoted to efficiently receiving, maintaining, and serving table data.

The second result is that an average 93% of bug reports stem from control plane errors. Our inspection of the code revealed two reasons for this. First, the control plane bugs tends to be complex–an artifact of the need to efficiently control the flow of large amounts of data. For example, 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 error that happens when clients concurrently access a migrating table.

	I/O Bytes Consumed/Generated		
Application	Control (%)	Data (%)	Total
CloudStore			
Master	100	0	252 MB
Slave	1	99	20.6 GB
Client	1	99	10.6 GB
Hypertable			
Master	100	0	120 MB
Metadata	100	0	228 MB
Slave	1	99	20.5 GB
Client	1	99	10.3 GB

Figure 2: Input/output (I/O) traffic size in bytes 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.

The second reason is that much of the control plane code tends to be new and written specifically for the unique and novel needs of the application. Code for table migration, for example, was *not* derived from an pre-existing and well-tested code base or library. By contrast, the data plane leverages previously developed code bases (e.g., libraries). For example, a large part of a Hypertable's data plane is devoted to compressing and decompressing a Hypertable cell-store. But most of this effort is undertaken by a well-tested library routine.

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 reads and writes to file descriptors (e.g., those connect to the tty, a file, a socket, or to 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 influenced/tainted by user data, then we considered it data plane I/O plane; otherwise it was treated as control plane I/O.

Results. Figure 2 gives the data rates for the control and data planes. At the high level, the results shows that the control plane is by far the least data intensive component. More specifically, the control plane code accounts for at most 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.

	(a) Code Size (Instructions)			(b) Reported Bugs		
Application	Control (%)	Data (%)	Total	Control (%)	Data (%)	Total
CloudStore						
Master	100	0		-	-	-
Slave	99	1		-	-	-
Client	99	1		-	-	-
Hypertable						
Master	100	0		100	0	5
Metadata	100	0		100	0	4
Slave	99	1		94	6	46
Client	99	1		5	0	20

Figure 1: Plane (a) code size and (b) bug count. As expected, the control plane accounts for almost all of the code and bugs in the datacenter application. CloudStore numbers are not given because it does not appear to have a bug report database.

4 Conclusion

A replay debugger for datacenter applications must reproduce distributed system errors 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 errors and that it operates at low data rates. Taken together, these results support our position that control plane determinism can enable practical datacenter replay.

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

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

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

[7] 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).

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

[9] 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).

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

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

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

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

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

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

[16] 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).

[17] 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).

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

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

[23] 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).

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