# Predicting Application Performance in the Cloud

by

Xuanran Zong

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Xiaowei Yang, Supervisor

_____
Jefferey S. Chase

_____
Shivnath Babu

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Computer Science
in the Graduate School of Duke University
2011

# Abstract

## Predicting Application Performance in the Cloud

by

Xuanran Zong

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Xiaowei Yang, Supervisor

_____
Jefferey S. Chase

_____
Shivnath Babu

An abstract of a thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science in the Graduate School of Duke University
2011

# Abstract

Despite the exceptional prominence of the cloud computing, the customers are lack of direct sense to select the cloud that delivers the best performance, due to the performance heterogeneity of each cloud provider. Existing solutions either migrate the application to each cloud and evaluate the performance individually, or benchmark each cloud along various dimensions and predict the overall performance of the application. However, the former incurs significant migration and configuration overhead, while the latter may suffer from coarse prediction accuracy.

This thesis introduces two systems to address this issue. CloudProphet predicts the web application performance by tracing and replaying the on-premise resource demand on the cloud machines. DTRCP further predicts the performance for general applications. In particular, it addresses the execution path divergence manifested during replaying the on-premise resource demand. Our experiment results show that both systems can accurately predict the application performance.

# Contents

# List of Tables

# List of Figures

# List of Abbreviations and Symbols

## Abbreviations

| | |
|---|---|
| DTRCP | Deterministic Trace and Replay CloudProphet |
| IaaS | Infrastructure-as-a-Service |
| PaaS | Platform-as-a-Service |
| EJB | Enterprise JavaBean |

# Acknowledgements

There are so many people I am indebted to.

Foremost, I would like to thank my adviser, Xiaowei Yang. Without her patient guidance that enlighten and encourage me to tackle each milestone throughout my graduate study, I would not be able to finish this thesis.

I also thank all my committee members, Jeff Chase, Shivnath Babu, and Bruce Maggs. Their comments and critiques during each of my milestone defense always inspire me with new ideas that further improve my thesis.

I am grateful to the main collaborators of my thesis, Ang Li, Srikanth Kandula, and Ming Zhang. Not only did they share so many fruitful ideas during the discussion, they also showed me how to be a good researcher.

I especially thank my lab mates, Qiang Cao and Xin Wu, who extensively discussed research problems with me. I also want to thank other people in my group, Yu Chen, Jennie Tan, Yang Chen. They brought so many interesting ideas during the weekly meeting that show me a broader view of other system research topics.

I express my gratitude to Marilyn Butler, who coordinates the graduate study program. Her responsive coordination always keeps me on track. She knows what the students really need and further helps the students at her best effort.

I want to thank Jiaoni Zhou and Xiao Huihui. They always stood by me and brought me laugh after one day of busy research. They kept me in peaceful mood after I was being blocked by each hurdle.

Lastly, I want to express my sincerely thank to my family. They encouraged me after each weekly phone call and gave me confidence to overcome the challenges.

# Chapter 1

# Introduction

## 1.1 Motivation and Challenges

Recent years have witnessed an increasing trend at cloud computing usage. The prominence of cloud computing come from its elasticity and the "pay-as-you-go" charging model. On one hand, with cloud computing, small enterprises pay no up-front investment to infrastructure and IT staff, reducing both the cost and the risk of over-provisioning; on the other hand, cloud providers can multiplex workload from many customers and improve the utilizations of their data centers.

Due to the significant economic incentives, many companies have started to offer public cloud computing services, such as Amazon EC2 [1], Google AppEngine [4], and Microsoft Azure [5]. A potential cloud customer considering migrating to the cloud hence needs to decide which provider is likely to offer the best performance to the application waiting for migration. One straightforward solution is to deploy the application on each provider and evaluate the performance accordingly. However, this incurs significant migration and deployment overhead, *e.g.* binary migration, static file migration, fulfilling library dependency, and etc. On the other hand, customers can also estimate application performance by leveraging the performance numbers published by cloud providers. However, for marketing reasons, cloud providers tend to be vague on the performance of their services. Although existing work [16] have

benchmarked each critical cloud service, such as storage, computation and network, the numbers provide indirect knowledge on the application's actual performance on different providers, due to the heterogeneous workload pattern between the application and the benchmarks. In an over-simplified case, an I/O benchmark results can barely predict the performance for a CPU-bound application. Given the complexity of the application in practice, it is almost impossible to find a benchmark sharing the same workload pattern.

In this thesis, we first introduce CloudProphet [3] that achieves accurate web application performance prediction by replaying the resource demand of an application on different cloud providers. Specifically, the system has two stages: a tracing stage and a replay stage. In the tracing stage, we instrument the application during its local execution to capture the application resource demand. In the replay stage, we replay the same demand on different clouds and use the performance number to resemble the application's true performance as if it has been migrated.

Compared to existing solutions, this basic technique has three advantages. First, it does not require actual migration, saving a large amount of configuration effort as well as licensing cost. Second, unlike previous solutions, our approach provides accurate application-specific performance estimation for each cloud provider. A customer can directly leverage this piece of information to choose the cloud provider with no domain knowledge. Further, the detailed performance number can even be useful in cost-performance analysis. Third, CloudProphet is fully automated that it only requires users to provide the application binary.

CloudProphet makes two assumptions on the web applications it targets at. First, different web requests are independent. This allows CloudProphet to accurately extract the true dependency for one request and only enforce the true ones during the replay. Hence, false dependency incurred by different request arrival order would not impose blocking overhead. Second, each request has deterministic resource demand. We

assume the application would always consume the same demand regardless of the cloud platforms.

The two assumptions are valid for most web applications, however, for general applications, they become arguable. On one hand, threads in general applications typically depend on each others, because programmers tends to use various synchronization primitives to ensure shared data integrity. On the other hand, some application logic depend on the shared data values so that different accessing order of the shared date can even change the resource demand. This could significantly skew the prediction accuracy, as the resource demand been issued is unfaithful. Further, given the platform configuration between the on-premise machine and the cloud instance could be dramatically different, the shared data accesses could have been out-of-order in high chance.

To address this issue, we further present DTRCP (Deterministic Tracing and Replay CloudProphet) that leverages deterministic replay to gradually infer cloud execution path by only running the application on the premise machines. We define execution path as resource demand together with their dependences, if any. Deterministic replay can steer the application to the correct execution path once it detects the cloud execution path diverges from the on-premise one. In this case, DTRCP assures to issue the faithful resource demand.

Our evaluation results show that both systems can accurately predict the application performance without migrating the application binary to the cloud. In particular, we show that CloudProphet can accurately predict the end-to-end response time for a real web application, RUBiS, with relative prediction error $< 20\%$. On the other hand, we also evaluated DTRCP's prediction accuracy for a non-deterministic application, UD Dropbox. The evaluation results show the effectiveness and importance of deterministic replay which is the key technique DTRCP used to gradually infer the faithful execution path.

Despite the accurate evaluation results, we observed that DTRCP is infeasible to the applications with a large number of concurrent threads. The problem is DTRCP requires the application to start over multiple times, and in the worst case, this number grows in factorial with the number of the concurrent threads. We propose some solutions to this challenge in the thesis and choose to leave it as a future work.

## 1.2 Organization

The rest of this paper will be organized as follows. We survey the related work in chapter 2. In chapter 3, we present the design of CloudProphet and some experiment results. This further motivates the design and implementation of DTRCP, shown in chapter 4. We finally conclude in chapter 5.

# Chapter 2

# Related Work

This thesis is the first attempt to predict application performance within the cloud context. The general performance prediction problem has been considered in many other contexts.

//Trace [20] predicts application performance on different I/O systems. It uses throttling to extract the true dependency between threads of an application, and then trace and replay the same I/O workload on different I/O systems while enforcing the true dependency. The technique assumes the application is deterministic: all I/O events remain the same even under throttling.

WebProphet [17] predicts web page loading time across different deployment platforms. It captures the dependency between various page components, *e.g.* HTML, CSS, JavaScript, and predicts how physical resource (*e.g.* CPU, network latency) upgrade would improve the performance of each page component, as well as how each page component loading time would affect the performance of others. WebProphet also leverages throttling to extract the true dependency among page components.

Previous studies have built models to predict the application performance under different load levels or deployment configurations. Stewart *et al.* built performance model for web applications to predict resource consumption (*e.g.* CPU, memory, and bandwidth) based on the web request arrival rate, burstiness, and request mix

rate. They further used the model to predict both throughput and latency as if the system was deployed using a different configuration.

Most performance prediction work make certain assumptions on the targeted applications. One key assumption is the applications should be deterministic in terms of the events being triggered. However, considering general applications nowadays, most of them are non-deterministic, manifesting distinct execution paths in different runs. The problem has already been well-studies, although not targeting at performance prediction, and deterministic replay is one classic solution that controls all sources of non-determinism during execution to preserve the same execution path.

Deterministic replay has been widely adopted by the debugging community [21, 22, 11, 13, 26, 23, 15], because programmers need to follow the same execution path in order to reproduce one bug. There are two sources of non-determinism: random inputs from the running environment (*e.g.* `gettimeofday`, `/dev/urandom`) and thread interleaving. While the former source can be easily controlled by recording and feeding the same inputs during replay, the latter one is much more difficult to control. One straightforward approach is to trace all the thread scheduling decisions and faithfully replay the same *physical* schedule. However, obtaining the physical schedule require kernel or hardware support, imposing a restriction on the platform. In addition, this approach may not suitable for multi-core machine, since it is non-trivial to acquire the thread interleaving given the threads are running in different physical cores concurrently [12]. Instead of obtaining the *physical* schedule, previous work propose to leverage *logic* schedule to control the non-determinism [11]. The key insight is shared variable access order can be also revealed by certain critical event (e.g. lock) order, assuming shared variable accesses are protected by critical events. Therefore, deterministic replay only needs to control the critical event ordering which can be obtained from local execution with modest overhead. We leverage similar idea in DTRCP that we only control the lock accessing order to reproduce the

cloud state up to the divergence.

# Chapter 3

# CloudProphet

## 3.1 Model and Assumptions

CloudProphet predicts end-to-end response time for web applications in the cloud. We only consider Infrastructure-as-a-Service (SaaS) cloud platforms, such as Amazon AWS and Rackspace. Customers tend to migrate their application to SaaS platforms because the same binary can be deployed seamlessly. In contrast, most Platform-as-a-Service (PaaS) cloud platforms only support a limited range of APIs, requiring the application to further refactor its implementation.

We model web application as a set of requests. Each request has a request processing path that traverses the components processing the request. Figure 3.1 shows one request processing path for a typical three-tier web application. The request traverses three components: a web server listening the web requests, an application server evaluating the servlets, and a database storing persistent data. The requests can be processed simultaneously, introducing resource contention on each component.

We further assume that there is no causal dependency between requests, because web requests are generally independent. The primary reason for making such assumption is that enforcing false dependency could dramatically skew the prediction accuracy. Considering the case that two requests are processed by the same thread

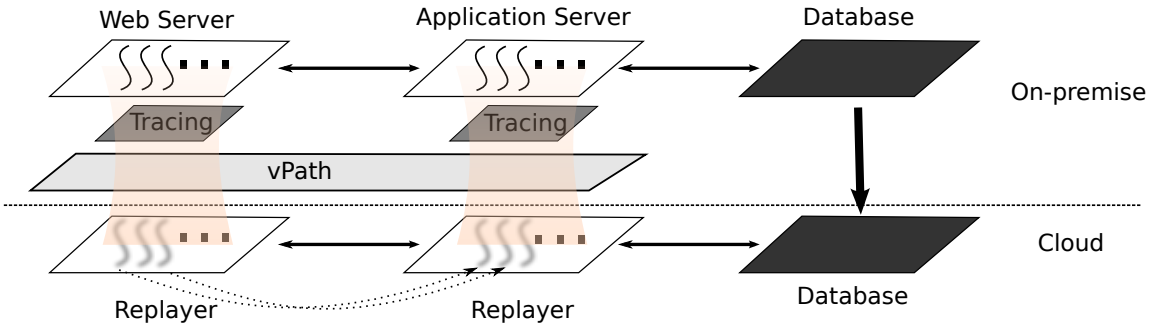


Figure 3.1: **An overview system diagram. The components communicate with each other. Each component runs multiple threads, which are traced and replayed in cloud. vPath algorithm can extract the thread dependency, which are shown in dotted arrow.**

sequentially during the tracing stage, this does not necessarily mean the two requests ever follow the same order. In the replay stage, the two requests can arrive out-of-order, because the replay runs on a different platform configurations. Enforcing the false dependency (a strict ordering between two requests) introduces dramatic overhead, as compared to the real case where the two request can be processed concurrently.

## 3.2 Design

CloudProphet leverages trace-and-replay to predict the end-to-end response time for web applications. As shown in figure 3.1, during the tracing stage, CloudProphet runs a tracing engine that traces all events triggered by the real execution. Each event denotes a piece of resource demanded by the application. The events are then feed into a dependency analyzer for extracting the request processing path. Finally, in the replay stage, the cloud replayer replays the same events (resource demands) on the cloud platforms.

Although trace-and-replay is a classic technique, we confronted three design challenges while we applying the technique in the web application performance prediction context. First, CloudProphet needs to trace a list of events capable of resembling the

resource demand of the application but not incurring too much tracing overhead. Second, CloudProphet needs to extract the request response path of each web request. Third, CloudProphet has to enforce only true dependency during the replay stage. In the rest of this section, we propose high-level solution to address each issue. For the detailed discussion, please refer to CloudProphet technical report [3].

### 3.2.1 Event Tracing and Replay

CloudProphet traces and replays four types of events.

**CPU Event** The CPU event represents CPU usage between two non-CPU events during the execution. The CPU events typically contribute a significant part of the overall resource consumption because web applications are more likely to spend CPU cycles to evaluate servelets. Moreover, web requests arrive simultaneously, often saturating the CPU capacity of the underlying platform. This introduces extra contention effect to the end-to-end response time.

Precisely tracing and replay the CPU events is hard. To achieve fine granularity, the CPU events should capture all types of CPU resources, such as the actual CPU instructions and memory footprints. Tracing such information is possible with low-level instrumentation, like PIN [18], but with huge($> 1000$x) run-time overhead. Therefore, the challenge is to strike a delicate balance between collecting the information necessary for prediction and incurring less run-time overhead.

CloudProphet uses the active running time between two non-CPU events to denote the CPU usage. To replay the same CPU usage, CloudProphet first scales up the CPU usage according to a pre-calibrated ratio, which is obtained through running a set of single-threaded benchmarks on both the cloud and on-premise platforms. The reason for scaling is the CPU usage depend not only on the application workload patter, but also on the hardware specification. CloudProphet then replays the scaled CPU usage by using a busy loop, where the loop iteration is determined by the

scaled CPU usage proportionally. This approach has two major advantages. First, active running time can be easily obtained through kernel interface `clock_gettime`. Second, although active running time does not reflect the actual CPU instructions and the memory footprints, and it is not representative across platforms, we found the scaling ratio achieve accurate prediction. The insight is there are only a few CPU vendors in today's market and most cloud providers offer homogeneous CPU to customers. Moreover, as the multi-core rapidly emerges recently, the CPU performance bottleneck is shifting from the single core performance to the concurrency issue, *e.g.* consistency and contention, across multiple cores. Tracing and replay the active running time can simulate the same contention effect and achieve reasonable prediction accuracy, as shown in section 3.4.

**Storage Event** There are two types of storage events: disk storage event and database storage event. Disk storage event represents a synchronized call that read/write data to the physical disk. The event denotes the actual disk I/O workload, so it needs to be faithfully replayed in the cloud. CloudProphet traces frequently used I/O calls from `libc`, such as `read()`, `write()`, and `fsync()`. To replay the same I/O workload, CloudProphet issues the same I/O calls. The difference is that the replay can operate on dummy data, saving the data migration cost.

Database storage event represents a database query issued to the back-end relational database system. We treat it different from disk storage event because customers tend to treat the database as an atomic entity. Tracing and replay database storage events is similar to the one for disk storage events. However, database storage events replay requires the actual database content. Fortunately, existing database data migration tools are mature enough to migrate the data automatically and seamlessly.

**Network Event** Network event represents a synchronized call that send/recv data

through the network. It denotes the actual network I/O workload that is another major component of the end-to-end response time for web applications. Tracing and replay network events are similar to the one for disk storage events. Note that faithfully replay the network events can reflect the actual network condition (bandwidth and latency) and how the network condition affects the end-to-end response time.

**Lock Event** Although web applications are less likely having locks across threads to speed up the performance, locks are inevitable under certain circumstances, such as when caching and logging are enabled. Lock event represents a lock primitive that ensures the mutual exclusion property for the application. So far CloudProphet only handles the basic locking primitives, such as pthread locks and file locks. It does not cover complex locks with built-in custom logic, because non-determinism matters in that case, and chapter 4 will discuss more on this issue. Due to the inherent simplicity of web applications, we did not find any complex lock in the applications we evaluated. Lastly, to replay the lock event, CloudProphet issues the same lock primitive.

### 3.2.2 Extracting Request Processing Path

Extracting correct request processing path is critical to prediction accuracy because it determines the severity of the false dependency. CloudProphet borrows vPath's [24] technique to address the issue. As shown in figure 3.1, vPath can analyze the event trace and extract the request processing path, namely how a single request is processed by threads in different components. In addition, vPath assumes a nested-RPC style communication model. Further, it assumes the application follows the dispather-worker threading model: each thread is dedicated to handle one request at any moment. Most server applications (*e.g.* Apache, JBoss) satisfies these two assumptions, while there are also exceptions (*e.g.* Nginx, Lighttpd). It is one of our on-going work to address server applications that follow the event-driven model.

### 3.2.3 Enforcing Dependency

Although web requests are mostly independent, causal dependency exist within one request. For instance, a request can only be processed after the client has sent it out, or the application server can only evaluate the servelet after the web server has received web request and sent nested requests to the application server. The request processing path encodes such causal dependency and CloudProphet enforces them during replay. To do so, a request processing path is further divided into event blocks according to the component where the event was traced. Given the nested-RPC style communication, an event block is triggered by its *parent* event traced on the predecessor component along the request processing path. During the replay, CloudProphet replays one event block only when its *parent* event replayed at other component has finished.

## 3.3 Implementation

We have implemented a CloudProphet prototype under Linux. The total line of code is around 7.5K. The tracing engine is implemented as a shim layer between the application and the `libc` library using LD_PRELOAD. It intercepts all library calls issued by the applications, including common disk I/O, network I/O, database queries, and locks. We also use btrace [2] to trace Java function calls, because some application queries database through Java libraries. We have also implemented vPath's algorithm to extract the request processing path. Lastly, the replayer is implemented as a loop that continuously issues the events as appeared in the trace and also enforces the causal dependency at the same time.

## 3.4 Evaluation

In this section, we first evaluate the single resource prediction accuracy by using a set of micro-benchmarks. Then, we evaluate the prediction accuracy for a full-fledged web application. Apart from the results we presented, the full evaluation report is in the technical report [3]. The following summarizes our main findings:

- CloudProphet can accurately predict the end-to-end response time for the web application we tested over a variety of cloud platform configurations under the normal workload level.
- Enforcing correct causal dependency is crucial to prediction accuracy.
- CloudProphet can accurately predict the time taken for each individual resource demand, *e.g.* CPU, disk I/O, etc.

### 3.4.1 Methodology

**Application** We present the prediction results for two types of applications. First, we use a micro-benchmarks suite, Phoronix CPU benchmarks [6], to evaluate the prediction accuracy for CPU resource. In particular, we choose nine single-threaded CPU-intensive benchmarks including multimedia encoding, photo processing, and cryptographic operations.

Besides the micro-benchmarks, we also evaluated the end-to-end response time prediction accuracy for RUBiS, a full-fledged web application. RUBiS [7] is an online auction website, followed after eBay. We choose RUBiS because it represents a typical three-tiered web application, which includes a web server to receive the requests, an application server to process the business logic, and a database to store the persistent data. In our experiment configuration, we use the official EJB version of RUBiS that runs on Tomcat, JBoss, and MySQL. For client, we use the default

| | Instance type | Database |
|---|---|---|
| Config 1 | AWS.m1.large | Xeround/AWS.m1.large |
| Config 2 | AWS.c1.xlarge | RDS (large) |
| Config 3 | Storm | Self-managed Storm |
| Config 4 | Rackspace | Self-managed Rackspace |

Table 3.1: **The four cloud deployment configurations used in CloudProphet's experiments.**

| | Provider | Instance | CPU freq. | Mem.(GB) |
|---|---|---|---|---|
| **Cloud** | Amazon | m1.large | 2 X 2.27GHz | 7.5 |
| | | m2.xlarge | 2 X 2.67GHz | 17.1 |
| | | c1.xlarge | 8 X 2.33GHz | 7 |
| | Rackspace | 2GB-tier | 4 X 2.2GHz | 2 |
| | Storm | 2GB-tier | 2 X 2.53GHz | 2 |
| **On-premise** | Deterlab | pc2133 | 4 X 2.13GHz | 4 |
| | Local | Optiplex745 | 2 X 2.4GHz | 2 |

Table 3.2: **Configurations of the cloud instances and the on-premise machines used in CloudProphet's experiments.**

workload generator bundled with RUBiS. Lastly, both Tomcat and JBoss satisfy the communication and threading model assumed by vPath.

**Cloud environment** Table 3.2 shows the cloud platform configurations we have used to evaluate the prediction accuracy for micro-benchmarks. In particular, we both used CloudProphet to predict the benchmark finishing time, and ported benchmarks and measured the real finishing time on each cloud platform for ground truth. We attempted to cover a diverse range of cloud platforms in terms of number of cores, memory size and their I/O (disk and network) speed.

To evaluate the prediction for the real web application, we built four candidate deployment configurations, as shown inr̃eftab:candidates. For each configuration, the web server and the application server were deployed on the same instance type; the database server was ran either on a cloud database or a self-managed database.

**On-premise environment**

To obtain an application trace, we need to run the application locally. For micro-

benchmarks, we ran them on our local machine. For the real web application, we built a small cluster on Deterlab and all machines have the same configuration. Both the configurations for the local machine and the deterlab machines are shown in 3.2.

### 3.4.2 Benchmark Study

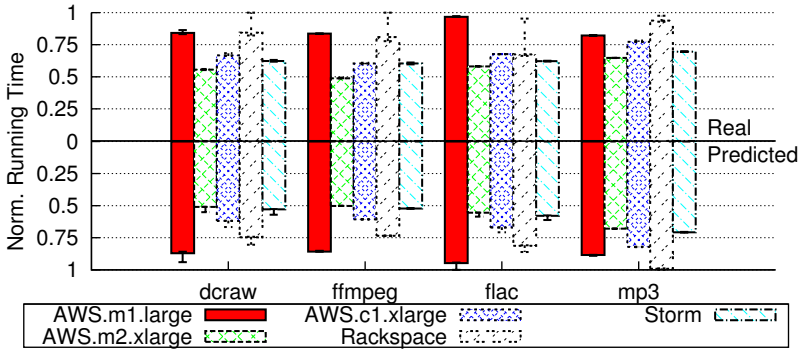


Figure 3.2: **The real benchmark running time and the predicted running time on five different instances. X-axis shows the benchmark, and the y-axis shows the normalized running time.**

Due to the space limit, we only show the CPU prediction results for single-threaded benchmarks, while the rest can be found in the technical report.

Figure 3.2 shows the prediction accuracy for four CPU benchmark tasks. The prediction experiment is repeated for 10 times. For each figure, the top half shows the normalized real running time and the bottom half shows the normalized predicted time. Each bar denotes the median time, as well as 5th/95th percentile, for one benchmark running on one cloud platform. We further clustered the bars according to the cloud platform for clarity.

The figure reveals that the scaling approach, although sounds straightforward, can accurately predict the CPU time. The relative prediction error for all benchmarks on all platforms are within 20%. The relative prediction error is calucated as $\frac{|t_{predicted}-t_{real}|}{t_{real}}$, where $t_{predicted}$ represents the predicted time and $t_{real}$ represents the real running time.

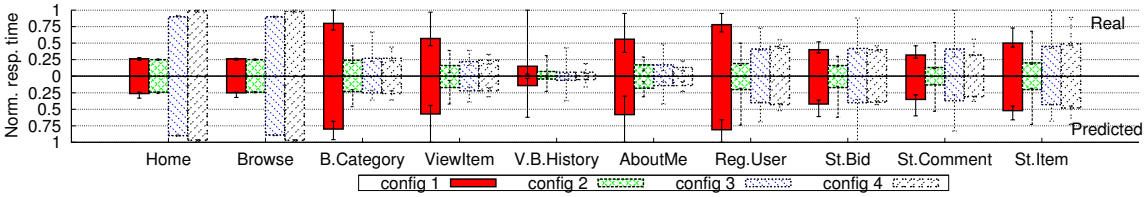


Figure 3.3: **The normalized response time of 10 RUBiS request types, shown along the x-axis, on four cloud migration configurations. The real and predicted time is shown in the top and bottom half of the figure. Time is normalized per cluster.**

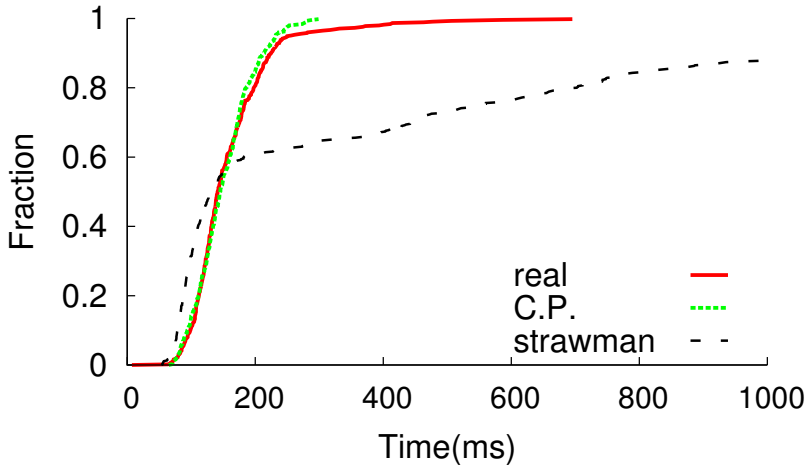


Figure 3.4: **The response time distribution of RUBiS' AboutMe request during the real run, the prediction run by** CloudProphet **(C.P.), and the prediction run by the strawman tool on config 2.**

### 3.4.3 Real Web Application

Figure 3.3 shows the end-to-end response time prediction accuracy for ten RUBiS page requests. Other page requests have similar results. The results are shown in the same format as figure 3.2. The experiment was conducted under normal workload level that the RUBiS client establisheused 20 concurrent user session to achieve around 50% average CPU utilization on both cloud and on-premise machines.

As shown in the figure, the relative prediction error for most pages are under 20%, except for V.B.History, which has a prediction error of 24%. This relies on two advantages of CloudProphet: first, CloudProphet can accurately predict the perfor-

mance for each resource, as shown in section 3.4.2; second, CloudProphet can extract the request processing path and mitigate the overhead caused by enforcing false dependencies. To show the latter advantage, we implemented a strawman CloudProphet with vPath disabled and all dependencies being enforced. We then compared the prediction accuracy provided by CloudProphet and the strawman CloudProphet. As shown in figure 3.4.3, about 40% of the end-to-end response time predicted by the strawman drastically diverge from the real response time. With further investigation, we conclude the large prediction error is caused by enforcing false dependencies. One server thread happened to process two requests sent by different threads during the tracing stage. However, during the replay stage, due to different CPU speed, the two requests were sent out-of-order and one request was blocked until the other one, which arrived late, had been processed. We discuss and address this problem in chapter 4.

# Chapter 4

# DTRCP

## 4.1 Problem Context

In the previous chapter, we show the importance of extracting and enforcing the correct dependencies. CloudProphet addresses this issue by targeting at web applications, whose request response path is deterministic and independent across requests. However, for more general applications, they expose non-deterministic behavior, meaning their execution path, which encodes the events and event ordering, would diverge significant if running on heterogeneous platforms. The execution path divergence may harm the performance prediction, because the event being replayed becomes unfaithful to the actual execution in cloud. This has two serious outcomes. First, the replay, if follows the unfaithful execution path, can lead to potential execution deadlock, as the execution path may violate the application's semantic meaning. For instance, programs using conditional variable typically leverage a shared flag to decide the invocation of `wait` and `signal`. If the replayer persist to issue the same event as in the trace and `signal` happens to be issued before `wait` during replay, the `wait` thread will be blocked forever.

Second, the replayer may issue imprecise events, because the divergence can steer the application logic that the real events become completely agnostic once the divergence occur. Figure 4.1(a) shows a code snippet demonstrating the non-determinism

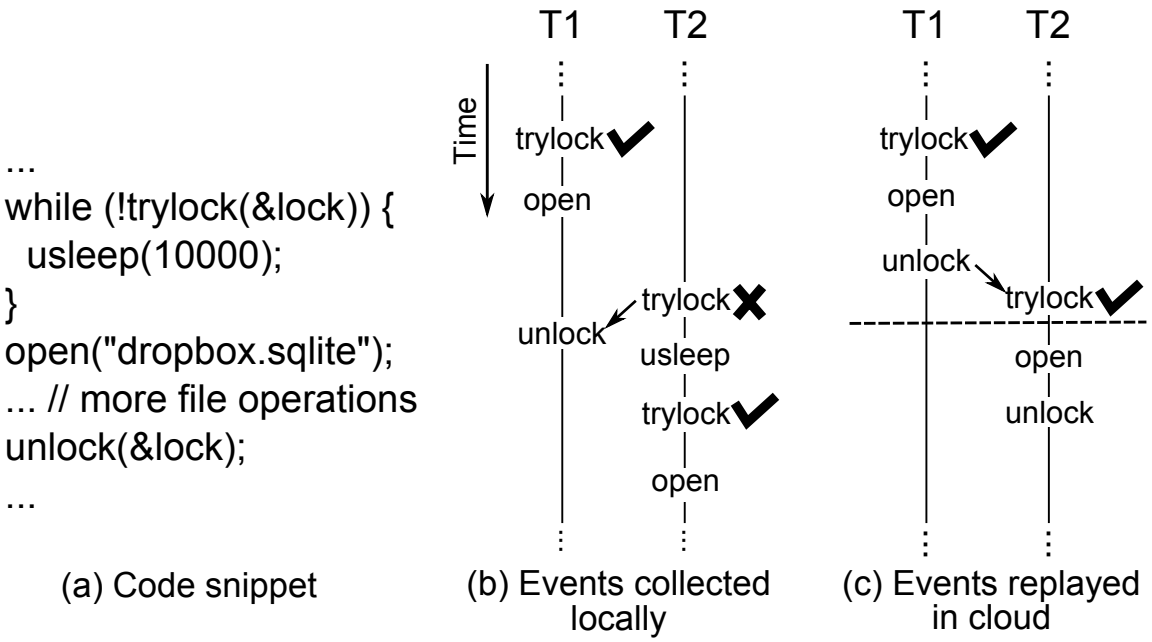


Figure 4.1: **(a) A code snippet that access a database file. (b) The events collected on the premise machine. (c) The events collected in the cloud. Note that the events are different from the on-premise ones.**

and the distorted events due to execution path divergence. The code attempts to acquire the lock and perform I/O (`open`). If the lock fails to acquire, the thread needs to pause 10ms. Figure 4.1(b) shows the on-premise execution path with two threads executing the same code. Given the slow on-premise I/O speed, thread 2 may fail to acquire the lock in its first attempt, and therefore triggers a `usleep`. However, since the cloud I/O is faster, thread 2 may success in its first attempt to acquire the lock, as shown in figure 4.1(c). In this scenario. the cloud execution path is different from the on-premise one. If we trivially replay the same events in the on-premise trace, the resource demand become unfaithful, as compared to the actual demand in cloud.

## 4.2 Application Model

DTRCP predicts the application finishing time for single box applications whose exe-

cution path may be non-deterministic. We model single box application as a group of threads all running on the same machine. Each thread contains a set of events. Each event corresponds to an actual library call executed by the application, such as `write` and `pthread_mutex_lock`, or an piece of CPU resource demand between two consecutive non-CPU events. A full taxonomy of captured events is given in section 4.3. An execution path of one application encodes the events and their orders traced in one particular execution. Note that for non-deterministic application, the execution path would different over runs.

After the thread issues one event during the execution, it may receive *external inputs* from the running environment that steer the execution to various control flows, eventually diverge the execution path. There are two types of external inputs: deterministic inputs and non-deterministic inputs. Deterministic inputs are static in different runs, hence they would not affect the execution path. For instance, application may have a static configuration file whose content, as read by one thread, would not alter in different runs. On the other hand, non-deterministic inputs could be different across runs and potentially alter the execution path. Since DTRCP targets at single box application, we consider two types of non-deterministic inputs. The first type of non-deterministic inputs is the return values from events that produce random inputs, such as pseudo-random number generator. The other type is the total execution order of the lock events which operate on the same lock, because it reveals the order of the shared variable accesses. We only consider lock event because this is the most widely used synchronization primitives. Further, other synchronization primitives, such as conditional variable, would only produce deterministic external inputs if we control the execution order of the lock events which protect the conditional variable [11].

## 4.3 Design

In this section, we first present the key observation that inspires DTRCP. Then, we show the high-level system design of DTRCP and the necessary assumptions. Lastly, we show the detailed design of each system component.

### 4.3.1 Challenges and Observation

The challenge is once the execution path diverges, we are completely agnostic about the events we need to issue with merely one on-premise execution trace. This could potentially lead to deadlock or/and imprecise prediction. Before we present our key observation, we first list some strawman solutions to address this problem and show why they are infeasible to solve the problem completely.

1. Deterministic replay [21, 22, 11] is one classic technique to reproduce the application behavior. It is mainly used by bug reproduction and is also suitable in our case. Borrowing this technique, we can control all non-deterministic inputs imposed on the system, such as random number and execution orders for certain critical events [11] during the replay stage, so that the replay is guaranteed to finish. In the previous conditional variable example discussed in 4.1, we enforce a strict order that `wait` issues before `signal`. Although deterministic replay addresses the deadlock issue, it introduces extra blocking overhead [10]. Further, imprecise prediction due to unfaithful replay still leaves unsolved.

2. Another solution is to consider each thread as independent, ignoring all synchronization and only replaying events that compose actual resource demand, such as CPU events and I/O events. The purpose of assuming this is to eliminate all non-determinism. However, this approach cannot guarantee deadlock free even with no synchronization, as blocking I/O could also freeze one thread. Moreover, eliminating synchronization could underestimate the performance,

because it overlooks the synchronization blocking time. Although potential models are available to mimic standard synchronization behavior, programmers tend build ad-hoc synchronization mechanisms, increasing the complexity to build a generic model.

Our key observation is that although deterministic replay may incur extra overhead during the replay, we can still leverage it during the tracing to gradually infer one correct cloud execution path. We argue that the execution path diverges due to some non-deterministic inputs have been received by the thread and steered the thread to a different control flow. In figure 4.1 example, for on-premise execution, thread 2's first `trylock` checks the lock state and receives the information that the lock is currently being hold by other thread; while for cloud execution, the same lock receives a distinct inputs notifying the lock is free to acquire. These pieces of information manifest themselves as *external inputs* to the thread that diverge the execution path. If we can capture all *external inputs* received in the cloud, we can deterministic replay the same cloud execution path on-premise and reproduce the cloud execution state up to the divergence point. Then, we can start to trace the events afterwards with no harness to explore the execution path after the divergence. Although the execution path is for on-premise machine, it will be the same as the one in the cloud up to the next divergence point. The reason is that the application starts from the same execution state, to which we apply deterministic replay to reproduce, and receives the same *external inputs* in both cases up to the next divergence point.

### 4.3.2 Overview

As shown in figure 4.3.2, DTRCP consists of four components: a tracing engine and a deterministic replayer on the premise machine, and an event issuer and a divergence handler on the cloud machine. To start the prediction, the user first runs the application on the premise machine with tracing enabled. The tracing engine

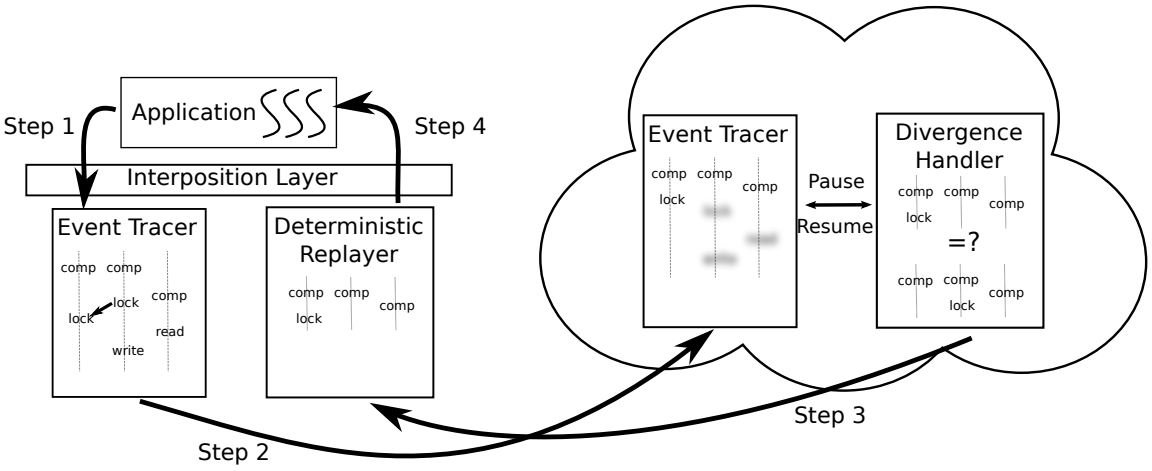


Figure 4.2: **System components and processing flow of** DTRCP**. Steps are labeled. Note that the flow takes iterative rounds, *i.e.* step 1 follows step 4, until the system terminates.**

interposes all events triggered by the application and constructs a trace with the on-premise execution path encoded (step 1). The tracing engine then sends the trace to the event issuer running on a cloud instance. The issuer's main task is to issue the same events to the cloud environment, simulating the same resource demands (step 2). In particular, the events are issued based on their ordering within the thread. Unlike deterministic replay, the issuer does not enforce any total order between events belonging to different threads, nor the return value from random sources. Therefore, as the non-determinism emerge, the actual cloud execution path would diverge from the on-premise one, which is the one being issued. For instance, the issuer may detect an out-of-order lock event or a random number generator that provides different external input to the thread. We refer the event that receives different external input as *divergence event*, the moment when the event receives the external input as *divergence point*, and the thread to which the event belongs as *divergence thread*.

At the divergence point, the divergence handler pauses the execution of all issuer threads, because the events afterwards become agnostic. It then constructs an issuer

trace containing the events that have been issued and the external inputs causing the divergence, and send the trace back to the on-premise machine (step 3). On the premise machine, the deterministic replayer deterministically replays the issuer trace up to the divergence point (step 4). In particular, it replays the same external inputs that causes the divergence on the cloud machine. By controlling all external inputs, the deterministic replayer is able to reproduce the cloud execution state at the divergence point. Beyond the divergence point, DTRCP removes the deterministic replay harness on the premise machine and let the application continue to run with only tracing enabled. This allows the on-premise tracing engine to collect a new execution path impacted by the external inputs from the cloud environment. With the new trace, DTRCP gain partial information of the cloud execution path after the current divergence point, but only till the next divergence point. The cloud issuer can start issuing the new trace until there is another different external inputs been received. Lastly, this approach guarantees incremental progress in the cloud, thus it can eventually terminate.

The correctness of DTRCP heavily relies on deterministic replay. We made two key assumptions that pose certain constraints on external inputs to ensure correctness of DTRCP:

1. There is no *hidden* communication channel that exchanges information between two distinct threads. In order to reduce overhead, we only interpose events at the library call layer, losing track of all shared variable accesses. We assume that shared variable accesses in most cases are protected by locks, whose order reveals the shared variable access order. We consider this a reasonable assumption, because shared variable without lock protection could eventually jeopardize the safety requirement of the application.

2. Applications are unlikely to use opaque system handlers to steer the control

flow. There are two types of opaque system handlers: virtual memory address and file descriptor. We consider this assumption reasonable because applications whose control flow depends on opaque system handler could produce random outputs, which are very rare in a well-written product.

The rest of this section shows a more detailed description of how each component functions.

### 4.3.3 Event Tracing

Unlike CloudProphet that only traces real resource demand, DTRCP also traces events that produces external inputs. In general, DTRCP traces one event if:

- it triggers real resource demand, *e.g.* CPU usage and disk I/O.
- it could block the execution, *e.g.* `sleep` and `waitpid`.
- it introduces non-deterministic external inputs, *e.g.* locks.

We select these three criteria because event satisfied one of these must play a significant role in the overall performance. In general, we trace five classes of events.

**CPU events** The CPU events are the same as the one in CloudProphet, which is discussed in 3.2.

**Disk I/O events** Since DTRCP targets at single box application and disk I/O typically contribute most I/O resource demand, DTRCP traces all disk IO events. Disk IO events include `read`, `write`, `open`, `fsync`, etc. Further, we also trace meta IO events, such as `lseek`, that modify the IO state and could potentially impact the performance.

**Process/thread management events** This class includes process/thread creation, termination and join events, such as `fork`, `waitpid`, `pthread_join`, etc.

DTRCP traces these events because they could block the execution that would seriously affect the application performance. However, these events typically do not introduce non-deterministic input, since their execution ordering are fixed in most cases.

**Timing events** Timing events are `sleep` and its variance. Although they do not introduce non-determinism, they block the execution and can easily become the performance bottleneck of the application.

**Synchronization events** There are two types synchronization primitives: lock and conditional variable. The lock introduces non-deterministic inputs because its order controls the logic, and the conditional variable can block the execution. Note that we do not consider conditional variable as a non-determinism source because it is typically protected by locks and its order strictly follows the lock order.

**Random events** This class of events produces random values, *e.g.* `rand` and `gettimeofday`, that could alter the execution path.

Note that for each event, DTRCP traces not only the event type, but also the associated argument values and the return values so the event issuer can issue the same event and the divergence handler can check whether the event yields a divergence.

### 4.3.4 Event Issuer

The event issuer issues events on the cloud machine. Since the issuer faithfully issues the process/thread management events, the issuer can maintain the same process/thread tree as the one manifested during the on-premise execution. Each issuer thread then look up the events belonging to the thread and issues them sequentially. Note that events can be issued concurrently, if they belongs to different threads.

DTRCP issues events in the same way as the one used by CloudProphet. Note that certain events contains opaque system handlers (*e.g.* file descriptor, process ID) and

the issuer may obtain different opaque system handlers as the ones in the trace. To handle this case, the issuer maintains a mapping between the system handlers in the trace and the ones obtained in the issuer run, therefore the issuer can look up the correct system handler to use when issues the event.

### 4.3.5 Divergence Handler

After the issuer issues one event, it needs to check if any divergence occurs. We define divergence event as the one that receives different external inputs from the ones in the trace. We consider two types of external inputs: the return values and the total execution order of the lock events. The issuer checks the return values for all events, because they may return different values if the application was running in a different platform. On the other hand, the total execution order is only specific to lock events, because the order reveals the shared variable access order that is crucial to the execution path. Thus, the issuer only maintains the total execution order for lock events that are operating on the same lock variable, and checks if any out-of-order happens.

When the issuer detects a divergence event, it becomes completely agnostic about the execution path after the divergence point. Therefore, the divergence handler needs to pause the execution and sends the partial trace that has been issued to the on-premise deterministic replayer in order to explore the execution path after the divergence point, as discussed in 4.3.6. Note that the partial trace, which we refer as issuer trace in the rest of this section, contains all external inputs received by the issuer threads to explore a faithful cloud execution path. The divergence handler resumes the issuer after the on-premise deterministic replayer has explored one faithful execution path and sent back to the issuer.

The divergence handler's job has one caveat. When it pauses and resumes the execution, the real wall-clock time is still ticking. Therefore, if an application's per-

formance heavily depend on the real wall-clock time, it would be seriously impacted by the length of the time period between pausing and resuming the execution. To address this issue, DTRCP maintains a virtual wall-clock time on the cloud machine. The virtual wall-clock time accumulate the time interval between pausing and resuming the execution, and deduct the accumulated time from the real wall-clock time when the application queries the wall-clock time. This technique simulates a 'world' (running environment) that between pausing and resuming the execution, both the 'space' (execution) and the 'time' (real wall-clock time) are temporarily stopped,

### 4.3.6 Deterministic Replayer

When the event tracer receives the issuer trace sent by the divergence handler, it leverages deterministic replay to reproduce the cloud execution state at the divergence point, in order to explore the faithful cloud execution path. In particular, DTRCP applies two rules to perform deterministic replay:

1. The deterministic replayer feeds the return values of the random events in the issuer trace to the local execution. This controls the first source of non-determinism resulting from acquiring random inputs from the running environment.

2. The deterministic replayer enforces a total order of execution for the lock events operating on the same lock. As shown in section 4.2, if we enforce the total order the lock events, we can control the non-deterministic inputs from all synchronization events. This controls the impact of the second source of non-determinism attributed by the thread interleaving.

After deterministic replaying the events in the issuer trace, DTRCP remove the deterministic replay harness and execute the application in the free run mode. However, as deterministic replay is intrusive to the execution, it may introduce different

amount of overhead to different threads. Therefore, threads can start to run freely at different moment. This can introduce unfaithful events in the local trace, because events executed in the free run mode may interact with the events still being deterministic replayed and further receives unfaithful external inputs from the events still running with the harness. For instance, a thread, which is in the free run mode, might execute non-blocking lock event (e.g. `pthread_mutex_trylock`) multiple times while the lock is still being hold by threads in the deterministic mode. On the contrary, if both threads have taken off the harness, the non-blocking lock would only be called once because the lock is freed after deterministic replay. The essential problem is the on-premise execution where threads take off the harness at different moment is inconsistent with the cloud execution where all threads are paused and resumed at the same moment. The inconsistency could lead to potential unfaithful execution paths. To address this problem, the deterministic replayer pause the execution of each thread at its first event entering the free run mode until all threads have accomplished deterministic replay. This guarantees the fidelity of the events been traced after the divergence point, because the execution is consistent with the one that could happen in the cloud.

## 4.4 Implementation

We implemented DTRCP on Linux and it supports most POSIX API calls as mentioned in section 4.3. Most of the tracing and replaying (event issuing) implementation is similar to CloudProphet, but with more events involved. Since DTRCP aims for performance prediction, we attempt to minimize the overhead and provide lightest intrusion to the real execution. Meanwhile, solutions capable of minimizing overhead may introduce extra technical challenges in other parts of the implementation. We present the main technical issues we have confronted. Lastly, we show how the

divergence handler pauses and resumes the event issuer.

### 4.4.1 Interposition Library

We built the event tracer with LD_PRELOAD that intercepts library calls called by the application. The reason we prefer LD_PRELOAD to other tools, such as `ptrace` and PIN [19], is that it imposed the lightest overhead on the actual execution. The interposition function for one event typically does two tasks: it executes the event and record it into shared memory where all recorded events are stored. Further, the deterministic replayer also relies on the interposition layer to control the return value and the lock order. Specifically, we use interprocess semaphore to enforce the execution order between two lock events.

Despite the light weight of LD_PRELOAD, it introduces extra technical challenges to the implementation. The main challenge comes from the tracing granularity. Since we are performing library layer tracing, we are unable to trace shared memory access. This is the key reason why we made the first assumption in section 4.3. By making the assumptions, we can instead trace the lock events.

Additionally, some events manifest complicated internal logic so that in order to perform deterministic replay, we need to trace the internal events. One example is `pthread_cond_wait`, which performs an internal lock operation at the end to ensure semantic integrity. However, as LD_PRELOAD interposes events at the library layer, it is impossible to intrude `pthread_cond_wait` to trace its internal lock. We address this issue by manually adding a pair of unlock and lock events at the end of `pthread_cond_wait`. Note that the execution after this modification would still manifest the same behavior because it is consistent with the `pthread_cond_wait` semantic meaning[1].

[1] `pthread_cond_wait` is not guaranteed to acquire the lock after it has been waked up

### 4.4.2 Event Size

In our original implementation, we used C union to store the argument values of all event types. Therefore, the event size is determined by the event with the largest number of argument values. The large event size introduced significant amount of communication and storage overhead. For example, it takes some time to insert a new event into the trace and the larger the event size is, the longer the insertion takes.

To address this problem, we obsoleted C union and built a customized event factory that allocates memory based on the event type. Most events only require small amount of memory, greatly reducing the memory usage and allocation overhead.

### 4.4.3 Pause and Resume the Issuer

The design of the divergence handler requires it to pause/resume all issuer threads at the same moment. However, users typically do not have privilege to access the virtual machine manager in the cloud. In our implementation, we approximate it by pausing each thread at slightly different moments, while still ensuring the fidelity of the events been issuing. In particular, the divergence thread coordinates the pausing/resuming operations. At the divergence point, other threads must be in one of the four states.

- The thread is emulating computation. In this state, the thread will repeatedly check if other threads have diverged, and if so, proactively pause execution of itself.

- The thread is issuing timing event. In this state, the divergence thread send a signal to pause the current thread. The timing event would not appear in the issuer trace, because it has not finished.

- The thread is issuing the event (except for timing event) that will not block the thread. In this case, the thread will pause itself after the event has finished. We consider the event valid and insert it into the issuer trace because it was issued prior to the divergence happened.

- The thread is issuing the event (except for timing event) that will block the current thread forever, because the divergence thread is paused. We distinguish this state from the previous state by employing a timeout counter ( 300ms). In this case, the divergence thread will send a signal to passively pause the current thread. Unlike previous state, we exclude the current blocking event from the issuer trace, because it has not returned.

To resume the issuer, the divergence thread sends a resume signal to all threads to resume the execution. Note that certain blocking events with timeout(e.g. `usleep` and `pthread_mutex_timedwait`) may require manual restart. We carefully calibrate the expired timeout period and deduct it when restart the event.

## 4.5 Evaluation

In this section, we show the prediction results of DTRCP. In particular, we focus on how deterministic replay can help inferring the cloud execution path and further improve the prediction accuracy. We omit the prediction results for each type of resource, because most of them are shown in section 3.4.

### 4.5.1 Methodology

**Application** We evaluated DTRCP prediction accuracy on UD Dropbox [9], an online file sharing website written in PHP and bundled with SQLite by default. When a user uploads a file, UD Dropbox records the meta-data, *e.g.* the user ID and the upload date, with the uploaded file and stores it into the SQlite database. SQLite

| | Provider | Instance | CPU model | Mem.(GB) |
|---|---|---|---|---|
| **Cloud** | Amazon | m1.large | 2 X Xeon E5507@2.27GHz | 7.5 |
| **On-premise** | Dep. cluster | linux27 | 16 X Intel Xeon E5540@2.53GHz | 48 |

Table 4.1: **Configurations of the on-premise machine and the cloud machine used in DTRCP's experiments.**

(version 2.0) [8] is a light-weighted file-based database. In SQLite, transactions are protected by a set of file locks, and if SQLite failed to acquire one lock, it will pause a short period and retry. At the end of the transaction, SQLite writes the data to the physical disk to ensure data consistency and integrity under unexpected shutdown. SQLite supports concurrent transaction, but given its lock-retry based transaction mechanism, other threads need to pause while one thread is holding the lock to process the transaction. The pausing time depends on the I/O speed of the machine. Hence, given different I/O speed between the on-premise and the cloud machine, the pausing time could be drastically different, producing different execution paths.

In the experiment, we built a client that uploads a 1MB file, and launched multiple clients simultaneously. We then measured and predicted the time taken for the client yielding the longest uploading time. Further, in order to show the effectiveness of DTRCP as compared to the approach with only trace and replay, we built a strawman issuer that issues event in cloud by exactly following the on-premise execution path via determinist replay. We also evaluated the prediction accuracy for the strawman.

**On-premise/Cloud environment** Table 4.1 shows the on-premise/cloud machine configuration we have been used in the experiment.

### 4.5.2 UD Dropbox

Figure 4.5.2 shows the predicted results of DTRCP match closely with the real processing time. We have evaluated DTRCP under three cases, each launches a distinct number of clients. For each case, we run the application 100 times to obtain the

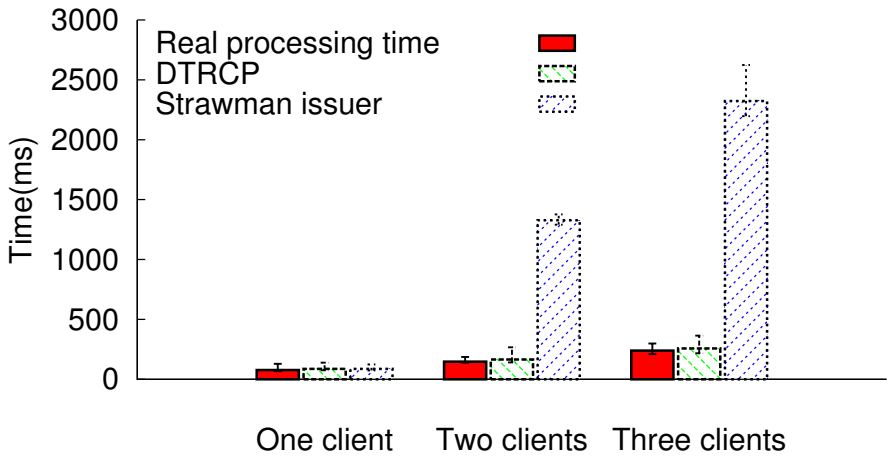


Figure 4.3: **Real/predicted processing time for UD Dropbox for distinct numbers of clients. Strawman results are also shown for comparison**

median, as well as the 5th/95th percentile, of the real/predicted processing time. As shown in the figure, the real processing time is almost the same as the time predicted by DTRCP in all three cases. The results indicate that DTRCP can accurately predict the resource demands, with or without resource contention. More importantly, the accuracy is not impacted by the execution path divergence. As the number of clients grows, the cloud execution path is more likely to diverge from the on-premise one, while the accuracy remain consistent across three cases.

To show the effectiveness of how DTRCP handles execution path divergence, we run the same experiment with a strawman issuer. As can be seen, the strawman achieves high accuracy when there is only one client. However, as the number of clients increases, execution path divergence emerges and the strawman behaves much worse because it issues unfaithful timeout events.

## 4.6 Limitations and Future Work

In the current design, the number of divergences depends on two key factors: the number of events that may produce non-deterministic inputs (N), and the number of

concurrent threads (K). In the worse case, each non-deterministic event may trigger divergences in $K!$ number of cases, thus the number of divergences equals to $N \times K!$. Moreover, DTRCP requires the application to start over when the divergence handler detects each divergence, in order to reproduce the cloud execution state. Therefore, as the number of threads grows, DTRCP becomes impractical to infer the faithful execution path.

We have confronted similar issue in some experiments. We have tested DTRCP using SPLASH2 [25] benchmark suite. Most SPLASH2 benchmarks are performing scientific computation and they share the same programming pattern: concurrent threads are synchronized through barrier and run in lock-step. In this case, DTRCP considers each barrier as a non-deterministic events. The experiment results show that some benchmarks have more than a hundred of barriers and DTRCP can barely finish with more than four concurrent threads.

DTRCP can be further studied and extended in the following two venues:

1. In the current design, as each divergence point requires one on-premise execution, DTRCP becomes infeasible to the application which has a huge number of divergence points. One optimization is to manually enforce the execution ordering for certain lock events. The key observation is that manually enforcing the execution order for lock events that are already in block state would not incur extra issuing overhead.

2. We plan to extend the targeted application from single-box application to distributed application. The challenge of handling distributed applications is it introduces an extra source of non-determinism: network latency. Network latency could potentially affect the execution path on either end of the machine. For example, one `recv` would receive less amount of data so the return value, now acting as a non-deterministic external input, would alter. In additional,

since distributed application lacks of a centralized controller, there are potential implementation challenges, such as to pause all the threads in different hosts. Fortunately, existing work [26, 14] have demonstrated the feasibility of deterministic replay for distributed system, hence our solution should remain sound.

# Chapter 5

# Conclusion

While cloud computing usage is ever growing these days, there is no easy solution for the customers to choose the cloud on which their applications perform the best. We present CloudProphet and DTRCP, both can predict the application's performance in the cloud prior to the actual migration. At current stage, CloudProphet shows promising prediction results for web applications by tracing and replaying the same resource usage on the cloud instances. On the other hand, DTRCP handles the execution path divergence by using deterministic replay and shows accurate prediction results for a non-deterministic application.

Despite the accurate evaluation results, DTRCP has its limitation that it can only handle applications with a few threads. As the number of threads increases, the number of divergences grows accordingly, making CloudProphet impractical to infer the faithful execution path. We leave this issue to the future work.

# Bibliography


[1] Amazon Web Service. http://aws.amazon.com.

[2] BTrace. http://kenai.com/projects/btrace.

[3] CloudProphet technical report. http://cs.duke.edu/~angl/cloudprophet_tr.pdf/.

[4] Google AppEngine. http://code.google.com/appengine.

[5] Microsoft Windows Azure. http://www.microsoft.com/windowsazure.

[6] Phoronix Test Suite. http://www.phoronix-test-suite.com.

[7] RUBiS: Rice University Bidding System. http://rubis.ow2.org/index.html.

[8] SQLite 2.0. http://www.sqlite.org/.

[9] University of Delaware Dropbox Services. https://pandora.nss.udel.edu/.

[10] D. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, page 57. ACM, 1994.

[11] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools - SPDT '98*, pages 48–59, New York, New York, USA, 1998. ACM Press.

[12] G. Dunlap and D. Lucchetti. Execution replay of multiprocessor virtual machines. *VEE*, page 121, 2008.

[13] Z. Guo, X. Wang, J. Tang, X. Liu, and Z. Xu. R2: An application-level kernel for record and replay. pages 193–208, 2008.

[14] R. Konuru, H. Srinivasan, and J. Choi. Deterministic replay of distributed java applications. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 219–227. IEEE Comput. Soc, 2000.

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

[16] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Internet Measurement Conference*, 2010.

[17] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. Webprophet: automating performance prediction for web services. In *NSDI'10*, pages 10–10. USENIX Association, 2010.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40:190–200, June 2005.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40:190–200, June 2005.

[20] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. //trace: parallel trace replay with approximate causal events. In *USENIX FAST*, 2007.

[21] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM ASPLOS*, 44(3):97–108, 2009.

[22] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192. ACM, 2009.

[23] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight rollback and deterministic replay extension for software debugging. In *USENIX Technical Conference*, 2004.

[24] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In *USENIX ATC*, 2009.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[26] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *USENIX NSDI*, number 1, pages 213–228. USENIX Association, 2009.