CloudProphet: Predicting Web Application Performance in the Cloud

Duke University Technical Report CS-2011-11

Ang Li Xuanran Zong Ming Zhang† Srikanth Kandula† Xiaowei Yang
Duke University †Microsoft Research
{angl, xrz, xwy}@cs.duke.edu {mzh, srikanth}@microsoft.com

Abstract – As public cloud computing services are gaining popularity, many are considering migrating their applications from on-premise to cloud. However, due to diverse cloud performance, choosing the cloud platform that is the best suited to migrate is a difficult unsolved problem. In this work, we present CloudProphet, a low cost tool to accurately predict the end-to-end response time of an on-premise web application if migrated to a cloud. CloudProphet collects a resource usage trace of the application running on-premise, and then replays it in cloud to predict performance. This approach does not require actual migration, and is agnostic to both the application and the cloud platform. We address two key design challenges in CloudProphet, including how to trace with both high fidelity and low overhead, and how to extract and enforce application dependency. Our evaluation shows that CloudProphet can achieve high prediction accuracy for two real applications on a variety of cloud instances from AWS, Rackspace, and Storm. We further show CloudProphet has low tracing overhead, and is much lighter than migrating the real applications.

1 Introduction

Public cloud computing has gained tremendous momentum lately. Major players, such as Amazon, Microsoft, and Rackspace, have been offering public cloud services for several years [2, 15, 18], while quite a few new players are expected to enter this increasingly competitive (and profitable) market soon [8, 10]. Riding this wave, an ever-growing number of business and personal customers alike either have migrated or are considering migrating their applications from on-premise infrastructure to the cloud to reduce IT expenses and improve scalability and availability.

This booming market has also led to a problem of plenty. Given various cloud providers, it becomes increasingly difficult for a customer to pick the best-suited cloud platform for her applications. Further complicating the matter, each cloud often offers multiple "tiers" or "plans" with different prices, configurations, and features. Questions like "should I go with Amazon's AWS or Rackspace's CloudServers?", and "is the large (xlarge) instance sufficient to handle my workload?" are frequent topics in public forums and mailing lists [7].

One of the key factors that influences a customer's decision in cloud selection is performance. Due to diverse hardware configurations and virtualization technologies, the performance of different cloud platforms can vary dramatically. One study [19] shows the disk I/O bandwidth of a virtual instance from one cloud can be twice as high as that of a similarly-priced instance from the other cloud. Another recent study finds that there is no single cloud which excels in all types of cloud service including computation, storage, and network [37]. For instance, a cloud that offers faster virtual instances may have lower inter-DC bandwidth. Since an application usually depends on multiple types of cloud service, it becomes really tricky to reason about the performance of an application across different cloud platforms.

A naïve approach to evaluate an application's performance on a cloud platform is to run the entire application on the target platform. In this way, the application owner can benchmark the application with representative workload, and accurately measure its performance. However, this simple approach can incur significant migration cost. First, an application usually depends on many third-party software and libraries, which also need to be set up in the cloud. A simple MediaWiki application (used in our evaluation) depends on software packages such as PHP, Apache, Lucene, Java, and XCache, while large business applications can be even more complex. Installing these packages can be tricky, due to licensing and compatibility issues. Second, moving the application itself can be time-consuming. Many applications have static components scattered around: configuration files, user data, shared libraries, etc. Migrating these static components tends to be tedious and error-prone. Finally, although cloud platforms usually support virtual machine images for fast startup, the images are not transferable across different platforms. Thus, the application owner has to setup the application on each of the cloud platforms which she intends to evaluate.

Realizing the problem above, we aim to develop a system, called CloudProphet, that can accurately predict an application's performance in a cloud without migrating the application. To build such a system, we must maintain a delicate balance between two conflicting goals. On the one hand, we want CloudProphet to be agnostic to both applications and cloud platforms, so that it is widely applicable to diverse applications and platforms. On the other hand, we want CloudProphet to be able to mimic the behavior of any target application with high fidelity, so that it attains high prediction accuracy.

One straightforward solution is benchmarking [6, 37].

If we can find a "representative" benchmarking task that exercises the performance bottleneck of an application, we can simply use the benchmarking results to quantify the application performance on different platforms. However, this works only for simple applications with one fixed bottleneck but not for most real applications with multiple bottlenecks. For instance, a web application typically uses CPU to run scripts, storage to record data, and network to communicate messages. It is almost impossible to find a "representative" benchmarking task since the actual bottlenecks could change under different workloads and platforms.

We present CloudProphet, the first system that can accurately predict the performance of web applications in the cloud. We focus on web applications because they are one of the most popular types of cloud applications and their behaviors are well understood. Our key idea is to use *resource usage trace* as a platform-independent abstraction of real application behaviors —- CloudProphet captures the resource usage of an on-premise application using a lightweight tracing engine, and then replays the resource usage trace on a target cloud platform to predict application's performance. The resource usage trace hides the application logic and platform details while allowing emulation of real application execution in the cloud.

A few technical challenges arise in developing CloudProphet. First, we need to trace application resource usage with the right level of details to attain both high fidelity and efficiency. Second, we need to capture application dependencies during tracing and enforce these dependencies during replay to faithfully emulate application behaviors under different platforms and workloads. The body of the paper discusses these challenges in detail and describes how we address them.

We have implemented CloudProphet and deployed it on three major public clouds: Amazon's AWS [2], Rackspace's CloudServers [18], and Storm [22]. Our evaluation shows that CloudProphet can accurately predict the benchmarking results of individual cloud services such as CPU, storage and network. By applying CloudProphet to two representative multi-tiered web applications (RUBiS [20] and MediaWiki [13]) with distinct resource usage patterns and bottlenecks, we find that CloudProphet can also predict the response times of these applications with low error rate ($< 10\%$ in almost all cases). We further demonstrate that application dependencies are vital to prediction accuracy and the tracing overhead has low impact on real application performance.

2 Problem Context

CloudProphet targets at predicting the performance of web applications when they are migrated to

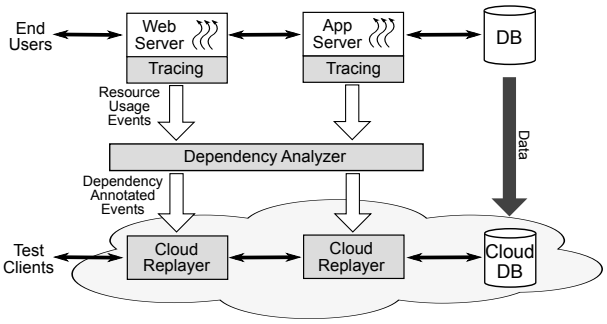


Figure 1: **The overall architecture of CloudProphet. The shaded boxes show the main components of CloudProphet: the tracing engine, the dependency analyzer, and the cloud replayers.**

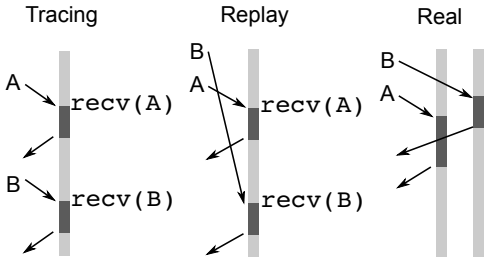


Figure 2: **An example to illustrate the false dependency problem. Each lightly shaded bar shows an event trace, and the dark segments correspond to the events that handle the incoming requests.**

Infrastructure-as-a-Service (IaaS) cloud platforms, such as Amazon's AWS and Rackspace's CloudServers. Today, many customers choose to migrate their on-premise applications to IaaS platforms because IaaS supports the binary of existing applications. In contrast, it takes much more effort to migrate applications to Platform-as-a-Service (PaaS) platforms, such as Google's AppEngine, because the applications have to be refactored using APIs specific to a target PaaS platform. Moreover, we focus on web application because it is one of the most popular type of applications running in the cloud [25] and its behavior is relatively simple and well understood. Designing a more generic prediction system for other types of application is beyond the scope of this paper.

Figure 1 illustrates how a typical three-tier web application works. Each user request may traverse multiple components and consume different amount of resource at each component. For instance, a search request may consume CPU at front-end server to execute PHP script and disk at application server to look up index files. A busy web application can serve many requests simultaneously. The end-to-end response time of a request is therefore not only determined by the amount of resource consumed by the request itself but also by the resource contention between concurrent requests.

Our goal is to predict the end-to-end response time of

web application running in the cloud. The response time is measured from the time when a request is sent by a user to the time when the corresponding response is received. CloudProphet makes prediction in three steps. First, it uses an online *tracing engine* to passively collect resource usage events from each component of a target on-premise application. Second, it uses an offline *dependency analyzer* to extract and annotate the dependencies between the events. Finally, it uses a number of *cloud replayers* to replay the events on a target cloud platform while complying with the dependencies. The replayers are driven by synthetic requests sent by one or more test clients simulating real users.

While the trace-and-replay idea has been adopted previously for performance evaluation [23, 28, 29, 32, 36, 41], we apply this idea to a new problem context which differs from existing work in two important aspects. First, CloudProphet predicts the *end-to-end* performance of an application which depends on multiple types of resource, while the existing work targets at fine-grained performance evaluation of one particular type of resource, such as I/O or network. Second, CloudProphet heavily leverages the dependency of web applications for performance prediction, while the existing work either assumes no dependency [23, 28, 32, 36] or assumes an entirely different dependency model, such as data dependency [41].

Several technical challenges remain. First, we need to determine the type of events to trace to be able to accurately predict application performance while being agnostic to application and platform. We also need to maintain a delicate balance between tracing granularity and system complexity. For instance, tracing each memory access [39] allows us to precisely reproduce the memory access pattern of an application. However, this would impose prohibitively high tracing and replay overhead.

Second, we need to correctly extract the dependencies between events in order to faithfully emulate application behavior in the cloud. A web application often serves many concurrent requests which may arrive in an arbitrary order. As illustrated in Figure 2, two requests, A and B, arrive in sequence during on-premise tracing run. During replay in the cloud, B happens to arrive before A due to network latency variation. If we naïvely replay the two requests according to the order in the trace, the replayer will hold B until A has been replayed. We call it a *false dependency* (since A and B are independent) which will lead to an over-estimate of B's response time. Replay should follow only *true dependencies* to produce legitimate response time estimate.

Third, we need to replay event trace according to inferred dependencies while incurring little extra overhead even at high request arrival rate. The replayer has to be highly optimized so that it will not skew response time measurement in the cloud.

In the current design of CloudProphet, we do not consider non-determinism in application logic, *e.g.*, the events triggered by a request are mostly stable regardless of when the request arrives or on which platform it is processed. This is true for the applications we studied. How to handle application non-determinism is out of the scope of this paper. The predictive capability of CloudProphet is also limited by the types of traced event. If an application is bottlenecked by an untraced resource type, *e.g.*, memory, prediction accuracy is likely to drop. We leave the extension of CloudProphet for memory-intensive applications as future work.

3 Design

The design of CloudProphet includes three major parts. First is to choose what resource usage events to trace and replay. Second is to accurately extract the application dependency. Third is to enforce dependency while replaying in the cloud. Below we describe each part in details.

3.1 Event Tracing and Replay

In this section, we describe the event types CloudProphet chooses to trace and replay. The chosen events cover the three common cloud resources used by web applications: CPU, storage, and network. CloudProphet further traces simple lock primitives to emulate lock contention. In the following, we describe each type of event in turn.

3.1.1 CPU Event

A CPU event captures an application thread's CPU usage between two non-CPU events. We choose the event because the CPU resource is commonly used by web applications to run the request-serving code. The running time can represent a significant part of the response time, especially for dynamically generated pages. Further, different cloud platforms have dramatically different CPU resources, such as different physical CPU models, different numbers of CPU cores, and different CPU sharing policies [18, 46]. These differences can greatly affect the application performance.

To precisely trace an application's CPU usage details, one would need to record every CPU instruction executed by the thread. However, this can bring prohibitively high tracing overhead, and is also unnecessary because CloudProphet's goal is to predict performance instead of accurately reproduce the application behavior. CloudProphet instead abstracts away the detailed instructions, and only records the *active running time* of the thread as a coarse approximation of its CPU usage. The active running time of a thread corresponds to the aggregate time when the thread is actively running on any of the CPU cores, and does not include the time when it is

waiting in the scheduler queue due to contention. Therefore, it shows the true CPU usage of a thread if running on a completely idle CPU core.

We choose to record the active running time for two reasons. First, the time is easy to obtain. Modern operating system kernels already track this information for scheduling and profiling purpose [9, 40], and can be easily read from user-space. For instance, under Linux the active running time of a thread can be obtained through the `clock_gettime` system call [5].

Second, although the active running time of a thread can change across different CPUs, we find that simply scaling the active running time traced on-premise is enough to predict the time in cloud with high accuracy (§ 5.2.1). This is likely due to the inherent similarity between the CPU types used by the major cloud providers. The scaling is based on a workload-independent ratio, called the single-thread performance ratio, which is calculated from CPU benchmark results. We measure the average active running time ratio among a number of single-thread benchmark tasks between the cloud instance and on-premise machine. In our experiments, we find the standard SPEC CPU2006 benchmarks [21] produce good prediction accuracy for a wide-range of workload.

CloudProphet uses a busy loop to replay the CPU resource usage recorded in a CPU event. Prior to replaying the CPU events, CloudProphet replayer first runs an one-time long busy loop in cloud to calibrate the ratio between the number of loop rounds and the active running time achieved. When replaying each CPU event, CloudProphet then uses the ratio to compute how many loop rounds are required to emulate the scaled active running time, and issues the busy loop without measuring any more active running time.

Choosing the active running time allows us to abstract away the complex details in CPU architectures and workload characteristics, and perform tracing and replaying at a high level. On the other hand, simplicity always comes at the cost. The main limitation of the technique is that it cannot accurately reproduce the contention of the low-level CPU resources, such as memory bandwidth and CPU caches. This is because CloudProphet does not capture the detailed usage information of those resources. Further, the scaling technique may introduce inaccuracy if applied between two very different CPUs. In this case, the scaling ratio may change significantly for different workload, and one scaling ratio may not fit all types of workload. Despite the limitations, we find the technique works well in practice for a wide-range of workload on real cloud instances, suggesting that the limitations are acceptable.

3.1.2 Storage Event

A storage event represents the application thread's access to the storage resources, including local disk and relational database. We choose the event because most web applications store their persistent state in the storage systems, and accessing the state can take non-trivial time. Further, a cloud platform may offer different storage options at different costs, and it is important to evaluate the performance under different storage configurations.

Disk storage event A disk storage event represents a synchronous system call to read and write data from/to the local disk storage. CloudProphet traces the events by intercepting the corresponding `libc` calls, such as `read()`, `write()`, and `fsync()`. For each event, CloudProphet records the data size as well as the context of the access to accurately characterize the resource usage. The context information includes a unique identifier of the target file (*e.g.*, its inode ID) and the file offset of the access. Note that the detailed data content is not recorded as it is not related to performance. Finally, CloudProphet also records the file size prior to the disk I/O call to help generate dummy files for replay.

Prior to replaying the storage events, CloudProphet needs to first create the dummy files to replay against. Each dummy file corresponds to a real file accessed by the application, and its initial size is the same as the size of the real file when it is first accessed. To replay a disk storage event, CloudProphet issues the same system call in the same context as recorded in the event. For instance, to replay an event that reads 450Bytes at offset 12354 from file 1, CloudProphet opens a pre-created dummy file that has the same size as file 1, seeks to the same offset, and reads the same number of bytes. To reduce the overhead of the control operations such as open and seek, CloudProphet caches an opened file descriptor per file per thread, and keeps track of the file pointer locations to avoid unnecessary seeking.

Database storage event CloudProphet treats relational database as a special type of storage resource offered by the cloud platform rather than an application component. This is because a cloud-based database usually shares the same interface as the on-premise one, but can have dramatically different implementations and configurations under the hood, making the resource usage trace collected on-premise unrepresentative. For instance, some cloud providers and third-party companies offer managed database services, such as Amazon's Relational Database Service (RDS) [1] and Xeround's MySQL services [26]. These services are likely to be different from the stock database systems due to custom optimization and tune-ups. Even if one decides to install the same database software in cloud as its on-premise one, it is very likely that she will need to re-tune the per-

formance parameters to fit the cloud instance's configuration, which changes the database's behavior.

A database storage event corresponds to a query sent to the database. CloudProphet traces the events by intercepting the library calls to the client-side database library (*e.g.*, `libmysqlclient`). For each event CloudProphet needs to record the full query content, because different queries can take different time to finish.

To replay the database events, the application owner first needs to migrate her database content to a cloud-based database. We believe the migration overhead is acceptable, because database dumping/importing is a fairly standard procedure, and many cloud providers, such as AWS and Rackspace, do not charge for incoming bandwidth [18, 25] in order to incentivize migration.

After the database migration, CloudProphet replays the events by re-sending the same queries to the cloud-based database. Similar to file descriptor caching, CloudProphet also caches an opened database connection per thread. After the thread has finished, the connection is returned to a global connection pool to be re-used by other threads.

3.1.3 Network Event

A network event represents a system call that sends or receives data to or from a network connection. We choose the event because network messages are used to synchronize between different application components and between the front-end component and the end users. The application's performance can also be affected by network bandwidth contention and long network latency.

Similar to disk storage events, CloudProphet traces the network events by intercepting the network `libc` calls such as `send`, `recv`, and `sendmsg`. For each network event, we record the number of bytes sent or received and the four-tuple that identifies the underlying connection: (`src_addr`, `src_port`, `dst_addr`, `dst_port`). We do not trace the connection setup and teardown calls such as `accept` and `connect` because the CloudProphet replayer manages the connections automatically in cloud (§ 3.3).

CloudProphet replays a network event by issuing the corresponding network system call to send or receive the same number of bytes along a connection chosen by the replayer. The CloudProphet design ensures that the replayer on the other side of the connection will always issue the matching network calls to avoid replay deadlock. We will describe the details in § 3.3.

3.1.4 Lock Event

The last type of event CloudProphet considers is lock. We choose the event because some applications use locks to ensure the consistency of the shared state between concurrent threads. For instance, the XCache extension of PHP uses file locks to protect the shared PHP opcode cache; many logging libraries also use locks to avoid log file corruption under concurrent access. Heavy lock contention might introduce additional blocking time to the application threads, which leads to performance penalty.

A lock event corresponds to a synchronous lock or unlock operation. CloudProphet traces the lock events by intercepting the common lock primitives, including pthread mutex and read-write locks, POSIX file locks, and Java monitor locks. For each lock event, CloudProphet records the operation (lock or unlock), the type of the lock (exclusive or shared), and a unique identifier of the lock object. CloudProphet abstracts away the implementation details of the different lock primitives as they are unimportant to reproduce the lock behavior.

CloudProphet replays the lock events by emulating the lock operations. For each lock event, CloudProphet issues a corresponding pthread lock call to lock or unlock a pre-created lock object and emulates the contention. Note that we do not replay using the exact same lock primitives as traced, because we focus on replaying the lock semantics instead of the detailed implementation.

So far CloudProphet only handles the basic locking logic such as exclusive and shared locks. We do not cover more complex locks (*e.g.*, read/write lock with intention), because replaying those locks would increase the complexity of the replayer, and it is unclear how popular those locks are in typical web applications. In our experiments with real web applications, we do not find such locks used by any of the application components.

3.2 Extracting Dependency

To reason about dependency, we consider a common nested-RPC style communication pattern. Such pattern is very popular in web applications due to its simplicity, and is also assumed by existing work such as [27, 43]. Under the assumption, the communication between two application components (or between the front-end component and the end-user) always happens through a pair of request and response messages sent over the same network connection. The receiver of the message serves the request by executing application logic and/or sending more *nested* requests to other components. It will eventually send back a response message to the sender after it has received every response from the nested requests. We further assume that the connection is not used for any other communication activity during the time between the request and the response.

Under such communication pattern, dependency in CloudProphet can be defined as the causal relationship between an incoming request and the resource usage events that serve the request. The events are referred to as an *event block*. For instance, when a web server receives an incoming request, its dependent event block includes a few network receive events to receive the request, a few

CPU events to run the scripts, and some network send events to send back the response. Many existing work can extract the same or similar dependency with a range of overhead and accuracy [27, 30, 33, 35, 43]. At one extreme, X-trace [33] requires a unique end-to-end ID to be embedded into each collected event through application annotation. From the IDs one can accurately associate the events with the requests, but annotating the application incurs significant overhead, making the whole low cost prediction scheme less attractive. At the other extreme, the "blackbox" approaches [27, 30] infer the dependency through statistical techniques. The approaches are application-agnostic, and only require the timing information of the events. However, inference error can affect CloudProphet's prediction accuracy and even cause replay deadlock. In the above web server example, if one send event is missing in the event block due to inference error, the client will deadlock as the numbers of bytes sent and received do not match.

In CloudProphet, we borrow the idea from vPath [43], and leverage the common threading patterns of web applications to accurately extract dependency while remaining agnostic to application. The key observation is that many servers follow a common dispatcher-worker threading model: a dispatcher thread accepts new incoming connections, and hands the connections to a number of worker threads, with each worker handling one connection. A worker thread then sequentially serves the one or multiple requests sent along its associated connection. We find the model is adopted by many popular multi-threaded servers such as Apache, Tomcat, and JBoss.

The threading model greatly simplifies dependency extraction. We use a high level description of the extraction algorithm to illustrate this. In the first step, all incoming requests at the front-end component are extracted from the network events. Second, for each request we leverage the communication pattern to find its corresponding response sent along the same connection. According to the threading model assumption, both the request and the response must be received or sent by the same worker thread. Moreover, the worker thread is only serving one request at a time. Therefore, all events happened on the worker thread between receiving the request and sending the response belong to the event block that depends on the incoming request. The algorithm further looks for nested requests sent within the event block, and recursively extracts the event block depending on each nested request. A unique ID is assigned to each request–event block pair for replaying.

Figure 3 shows a pseudo-code of the extraction algorithm. The algorithm includes four steps. Line 1-10 group the network events by connections. A connection is identified by the four-tuple recorded for each network

Input: $Events_c$/$Events_t$, the event trace for application component c/thread t; C, the set of all components; c_{front}, the front-end component.

```
1:  for c ∈ C do
2:      sort_time(Events_c)
3:      for e ∈ Events_c do
4:          if is_network(e) then
5:              conn ⇐ e.conn
6:              Conn_events_conn.add(e)
7:              Conn_set.add(conn)
8:          end if
9:      end for
10: end for
11: for conn ∈ Conn_set do
12:     Msgs_conn ⇐ extract_msgs(Conn_events_conn)
13: end for
14: for conn ∈ Conn_set do
15:     conn_r ⇐ find_reverse_conn(Conn_set, conn)
16:     match_msgs(Msgs_conn, Msgs_conn_r)
17: end for
18: for conn ∈ Conn_set do
19:     if !conn ∈ c_front then
20:         continue
21:     end if
22:     for msg ∈ Msgs_conn do
23:         if !is_incoming(msg) then
24:             continue
25:         end if
26:         msg_resp ⇐ get_response(msg)
27:         first_e ⇐ get_first_event(msg)
28:         last_e ⇐ get_last_event(msg_resp)
29:         t ⇐ get_thread(first_e)
30:         bid ⇐ new_block_ID()
31:         msg.ID ⇐ bid
32:         block_bid ⇐ get_events_inbetween(Events_t, first_e, last_e)
33:         Recursive_msgs ⇐ get_outgoing_msgs(block_bid)
34:         for out_msg ∈ Recursive_msgs do
35:             match_msg ⇐ find_match_msg(out_msg)
36:             find_event_block(match_msg)
37:         end for
38:     end for
39: end for
```

Figure 3: The pseudo-code to extract the dependency between a request and an event block.

event (§ 3.1.3). Note that the two ends of a connection are treated as different connections at this step. Based on the communication pattern, line 11-13 extract the network messages from each connection's events, by grouping the consecutive send or receive events. line 14-17 match the outgoing and incoming messages at two different ends of the same connection. Finally, line 18-39 show the recursive algorithm that extracts the dependency.

The main limitation of the above algorithm is that it cannot handle the event-driven threading model. Under such model, each worker thread can serve multiple

requests simultaneously by "context-switching" between requests when they are blocked by network I/O. Further, a request can also be handled by different workers at different stages. We leave this as our future work.

3.3 Enforcing Dependency

A cloud replayer enforces the dependency by replaying an event block only when its depending request has arrived. Similar to a multi-threaded server, a cloud replayer also has one dispatcher thread that listens for incoming requests, and several worker threads to replay the event blocks. Once an incoming request arrives, the dispatcher looks up the corresponding event block from the trace, and assigns one worker thread to sequentially replay the events in the block.

Two design issues need to be addressed here. First is how to decide which event block to replay. We require the sender of a request to first send the request's event block ID prior to actually sending the request, so that the dispatcher thread at the receiver can choose the right event block to replay. Second is how to manage the connections. CloudProphet pools the connections between a pair of replayers to emulate the persistent connection feature supported by some protocols (*e.g.*, HTTP/1.1). The maximum size of the connection pool is adjustable. It can also be set to zero to emulate the connection setup overhead for legacy applications.

One or more test clients drive the replayers by sending synthetic requests to the front-end replayer and measure the predicted response time. The requests are derived from the front-end component's event trace. A test client can work in two different modes: in the *independent* mode, the requests are sent out at exactly the same pace as seen in the trace. This ensures that the front-end replayer sees the same request arrival rate as the original front-end component, and is suitable for the case where the requests are independent from each other. In the *session* mode, the requests are grouped into multiple sessions. While different sessions are replayed independently, the "think time" between two consecutive requests within a session is preserved. This mode is tailored for session-based applications, where the request arrival rate also depends on application performance. By default, CloudProphet works in the independent mode, but if the session information is available, CloudProphet can also take it into account and improve prediction accuracy.

4 Implementation

We have implemented a CloudProphet prototype under Linux including its three main components: tracing engine, dependency analyzer, and cloud replayer. The total line of code is around 7.5K.

The tracing engine is implemented as a library call interceptor using the LD_PRELOAD technique. We choose the technique because it can trace the `libc` as well as the native database library calls without instrumenting the application. We also use btrace [4] to trace Java function calls transparently. All tracing state is stored in shared memory. In its current form, the engine can trace the common disk I/O, network, and locking calls in `libc`, the Monitor enter/exit in Java, and the API calls to the MySQL client library in both C and Java. We also implement the dependency analyzer according to the algorithm in § 3.2, and the cloud replayer as a lightweight multi-threaded server. In the following, we describe two key implementation challenges we address.

Reducing memory management overhead In our tracing engine implementation, we keep an in-memory buffer for the traced events and periodically flush the events to disk. In our original design, we use the stock memory management functions (`malloc` and `free`) to manage the buffer: an event structure is allocated or freed whenever a new event is traced or an existing one is flushed away. However, the functions introduce non-trivial overhead when the event arrival rate is high. We then implement a custom memory management scheme using an efficient circular buffer. The buffer includes a ring of 1MB buffer blocks. A new block is added if the existing ones are full, and its allocation overhead is amortized over all the events in the new block. Further, because of the circular structure, the flushed buffer blocks are automatically reused. In our evaluation, we find that the new memory management scheme can reduce the tracing overhead by up to 50% (§ 5.4).

Accurate active running time measurement in cloud During our experiments on real cloud instances, we find that some instances have CPU sharing enabled and a significant portion of the CPU time is stolen by other co-located instances (§ 5.2.1). This feature interferes with the active running time accounting, because the system call `clock_gettime` does not exclude the stolen time from the accounting result. The problem can affect the scaling ratio computation and the replay calibration, because CloudProphet needs to measure the active running time of a benchmark task or a calibration loop in cloud (§ 3.1.1). On the other hand, the current Linux kernel only keeps stolen CPU time statistics on a per CPU core basis. To resolve this issue, we set the affinity of the benchmark or calibration task to only one CPU core, and collect the active running time of the task as well as the stolen CPU time on the core. Because both the benchmarking and the calibration are done prior to replay when the instance is idle, it is likely that all stolen time on the core is included in the active running time. Therefore, we can subtract away the stolen time to obtain the real active running time of the task. The tech-

Resource	Benchmarks
CPU	Phoronix's CPU benchmarks [16]
Disk storage	IOzone [11]
Database storage	TPC-C [24]
Network	File downloading
Lock	Custom webapp using log4j [12]

Table 1: **The benchmarks we choose to evaluate the prediction accuracy for each resource type.**

nique helps CloudProphet to accurately predict the running time of CPU-intensive workload regardless of CPU time stealing (§ 5.2.1).

5 Evaluation

We evaluate CloudProphet along three different dimensions. First, we use a set of benchmark applications to test CloudProphet's prediction accuracy for each resource type. Second, we use two representative multi-tiered web applications to evaluate CloudProphet's effectiveness in predicting the end-to-end response time of real applications. Third, we evaluate the various overheads introduced by CloudProphet. The following summarizes our main findings:

- CloudProphet can accurately predict the end-to-end response time of two real applications over a variety of cloud configurations under a range of normal load levels.
- Dependency extraction is critical to prediction accuracy. Without extracting the correct dependency the prediction accuracy degrades significantly due to false dependency.
- CloudProphet can accurately predict the benchmark results of CPU, storage, network, and lock. Due to space limit, we only show the CPU benchmark prediction results.
- The tracing overhead of CloudProphet only increases the application's response time by 7% and reduces its maximum throughput by 11%.

5.1 Methodology

Applications In our evaluation, we first choose a set of simple benchmarks targeted at each specific resource type to evaluate CloudProphet's prediction accuracy along each resource. Table 1 summarizes the benchmarks. For CPU, we choose nine single-thread CPU-intensive benchmark tasks from the phoronix test suite, including multimedia encoding, photo processing, and cryptographic operations. The tasks have different characteristics. For instance, the maximum resident set size (RSSes) ranges from 6MB to 324MB, and the instructions-per-cycle (IPC) on our local machine ranges from 1.371 to 2.164. For storage, we use the standard benchmarks such as IOzone and TPC-C. For network, we set up a simple website serving static objects, and predict the time to download in both bandwidth-bound and latency-bound cases. For lock, we use a custom Java web application where each request triggers heavy disk write to a lock-protected log file through the log4j logging library. We then send the requests at a high concurrency level, and predict the response time of each request, which includes both the disk I/O time and the time to wait for the lock.

Besides benchmarks, we also choose two representative web applications, RUBiS and MediaWiki, to evaluate CloudProphet's end-to-end prediction accuracy. RUBiS [20] is an online auction application modeled after eBay. We choose RUBiS because it represents a typical three-tiered web application, which includes a web server to render the web pages, an application server to execute business logic, and a database to store application data. We use the official Enterprise JavaBeans (EJB) implementation of RUBiS that runs on Tomcat, JBoss, and MySQL. We also use the official workload generator to generate representative user requests. Both Tomcat and JBoss satisfy the threading model in § 3.2.

MediaWiki [13] is a popular PHP-based Wiki software used to host Wikipedia. We use a search-enabled MediaWiki distribution which includes a web server, a dedicated full-text search server, and a database. The search server maintains an index of the database content, and answers search queries sent by the web server. We choose such distribution because it represents a typical information sharing web application, and the search function represents standard search engine workload. We choose the common MediaWiki deployment that includes Apache (with PHP5 module) as web server, the official MediaWiki search engine [14] as search server, and MySQL as database. We use a sampled dataset from Wikipedia (>300K entries) as our test data. Both Apache and the search engine satisfy our threading model.

Because there is no automatic workload generator for MediaWiki, we generate a synthetic trace of requests and replay it at different speeds. The trace includes real Wikipedia page read (Get) and update (Post) requests from its access log [44]. We also generate search requests using the top 100 popular English words as keywords to ensure the existence of search results. In the end, we randomly mix the requests together to generate the synthetic trace. We have tested different request mixes and found the mix ratios have no impact on prediction accuracy. In the results shown in this paper, we use a mix of 70% page read, 20% search, and 10% page update.

Cloud environment In our experiments, we try to predict the application performance in three popular IaaS cloud platforms: Amazon AWS [2], Rackspace CloudServers [18], and Storm [22]. Each platform offers multiple tiers of instances and database services.

Due to cost reason, we cannot cover every instance

	Provider	Instance type	CPU model	Memory (GB)	Price ($/hr)
Cloud	Amazon	m1.large	2 X Intel Xeon E5507@2.27GHz	7.5	0.34
		m2.xlarge	2 X Intel Xeon X5550@2.67GHz	17.1	0.50
		c1.xlarge	8 X Intel Xeon E5410@2.33GHz	7	0.68
	Rackspace	2GB-tier	4 X AMD Opteron 2374 HE@2.2GHz	2	0.12
	Storm	2GB-tier	2 X Intel Xeon X3440@2.53GHz	2	0.07
On-premise	Deterlab	pc2133	4 X Intel Xeon X3210@2.13GHz	4	N/A
	Local	Optiplex745	2 X Intel Core 2 6600@2.4GHz	2	N/A

Table 3: **Configurations of the cloud instances and the on-premise machines used in our experiments.**

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

Table 2: **The four cloud deployment configurations used in our experiments.**

type and database service. Therefore, we strategically choose a few representative instances and services to achieve both diversity and comprehensiveness. Table 3 shows the configurations of the five cloud instance types used in the experiments. For diversity, we try to cover as many product lines as possible, because the instances from the same product line are likely to share similar physical hardwares and therefore have similar performance characteristics [3, 37]. The five instance types we choose cover all three product lines of Amazon, and the only product line for Rackspace and Storm respectively. We find the CPU models of the instances are all different, and cover the common micro-architectures such as Intel's Penryn and Nehalem, and AMD's Shanghai. For comprehensiveness, we try to cover instances with different resource levels, by selecting instances at different pricing tiers. The hourly prices of the instances range from $0.07 to $0.68. Their resource levels also differ significantly. For instance, the high-end instance has eight CPU cores, while the low-end one only has two.

In terms of database, we test both managed and unmanaged services. For managed services, we choose both Amazon's RDS and Xeround, a recent startup which offers auto-scaling MySQL service inside both Amazon and Rackspace's cloud. For unmanaged services, we install MySQL on a dedicated instance of the above five types, and tune the performance parameters according to MySQL's example configuration files.

To predict the performance of real web applications, we further choose four cloud configurations as candidate deployment scenarios for the applications. Table 2 shows the configurations. Each configuration has one instance type to run the application components, and one database service to store the application data. The configurations cover four out of the five instance types in Table 3, and the database services cover both managed and unmanaged services. Because Xeround in config 1 does not support blob index required by MediaWiki, we replace it with an unmanaged MySQL running on AWS' m1.large instance for MediaWiki. Lastly, we ran the clients of the web applications on our local machines at Duke University.

On-premise environment To obtain the resource usage traces, we also need to run the applications in an on-premise environment. We choose two physical clusters for this purpose. One is a small local cluster inside our department for the benchmark study, and the other is a larger one inside Deterlab [31] for real application prediction. The configurations of the cluster machines are shown in Table 3.

5.2 Benchmark Study

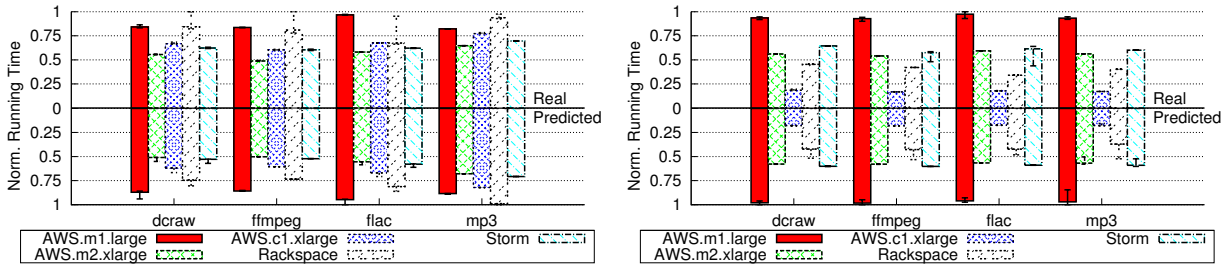
In this section, we evaluate CloudProphet's prediction accuracy for the benchmark results.

5.2.1 CPU Prediction

Figure 4(a) shows the prediction results of four CPU benchmark tasks. The results of the other tasks are similar and omitted. The predication experiment is repeated for 10 times. The top half of the figure shows the real running time of the tasks on different cloud instances, while the bottom half shows the predicted time. Each bar shows the real or predicted time of one task running on one instance type, and the error bar shows the 5th and 95th-percentile values. The bars are clustered by tasks, and for clarity , we normalize the bars within each cluster.

The figure shows the prediction accuracy of every task on every cloud instance is high, as the bottom half almost mirrors the top half. We further compute the relative prediction error as $\frac{|t_{predict}-t_{real}|}{t_{real}}$, where $t_{predict}$ and t_{real} are the median predicted and real running time, respectively. The results show that the relative prediction error of all tasks and instance types is below 20%.

We find there are two key factors critical to the high accuracy. First, the performance difference between different CPU models does not depend strongly on the workload. As can be seen in the figure, the relative height difference between the bars in one cluster is similar across the four heterogeneous tasks. There is only one exception: AWS.m1.large, which we will explain later. We speculate that the comparable running time is due to the



(a) Single-threaded

(b) Multi-threaded

Figure 4: **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.**

inherent similarity between different CPU models, especially the ones from the same vendor. This observation explains why CloudProphet can achieve good prediction accuracy by just using the same scaling ratio across all workload types.

The second factor is that CloudProphet can accurately emulate the CPU stealing effect in cloud. From the figure we can see the instance AWS.m1.large has different performance across different tasks. It performs much slower than AWS.c1.xlarge for the first three tasks, while the two have almost the same performance for the last task `mp3`. Detailed analysis shows that the low performance of AWS.m1.large is actually caused by CPU stealing. Using the technique in § 4, we find that around 30% of the running time of the first three tasks is actually stolen time. On the other hand, `mp3` receives minimum impact from CPU stealing, possibly because it is very lightweight (only takes ∼100ms to finish), and the CPU stealing policy is load-dependent. In this case, if we naïvely compute the scaling ratio without considering the stolen time, we may end up over or under-estimating the real performance of the instance, depending on which benchmark we use. CloudProphet addresses this issue by always measuring the real active running time in cloud, excluding all stolen CPU time. In this way, CloudProphet can reproduce the same physical CPU usage as the real application, and emulate the effect of CPU stealing.

We further test CloudProphet's prediction accuracy under CPU contention. To do so, we run multiple instances of the same benchmark task concurrently, and predict their running time on the cloud instances. Figure 4(b) shows the prediction results with eight concurrent instances. The results under other concurrency levels are similar. Due to CPU contention, the running time of the tasks differs significantly across different instances, because of the different number of cores. CloudProphet again can accurately predict the running time, because the tracing engine can faithfully extract the active running time of each benchmark thread by using the kernel statistics, even under heavy contention. At the same time, the replayer can replay the CPU usage of each thread separately, and emulate the contention effect. The result also suggests that, even under heavy contention, the benchmark tasks are unlikely to be bottlenecked by other CPU-related resources, such as memory bandwidth and cache. Therefore, by emulating CPU contention alone CloudProphet can achieve good prediction accuracy.

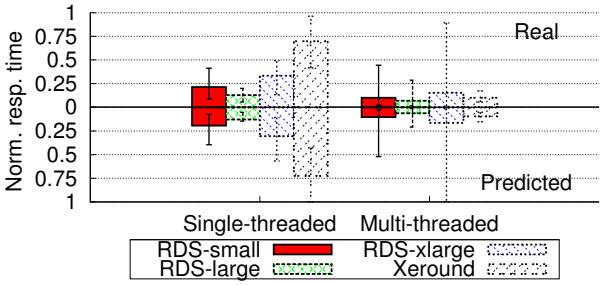


Figure 6: **The real TPC-C transaction time and the predicted one on three AWS RDS managed databases and one Xeround managed database. X-axis shows the two test scenarios: single-thread and multi-thread. Y-axis shows the normalized transaction time.**

5.2.2 Storage Prediction

Disk storage Figure 5 shows the real and predicted running time of IOzone. We test all eight present I/O patterns with 128MB files and 4KB records. The results for other file and record sizes are similar. As can be seen, the predicted values closely match the real ones under different I/O patterns and across all five instance types. The relative prediction error is below 20% in all cases. Some cloud instances have significant performance variation under certain I/O patterns, such as Rackspace's instance under "sequential write" and "random read/write". We suspect this is due to the cloud's internal interference. CloudProphet can faithfully re-produce the variation, as the predicted error bars also match the real ones.

Database storage Figure 6 shows the predicted and real transaction time of TPC-C. We test both a single-thread case with one client sending transactions in sequence, and a multi-threaded case with 30 clients running concurrently. We only show the results with managed cloud

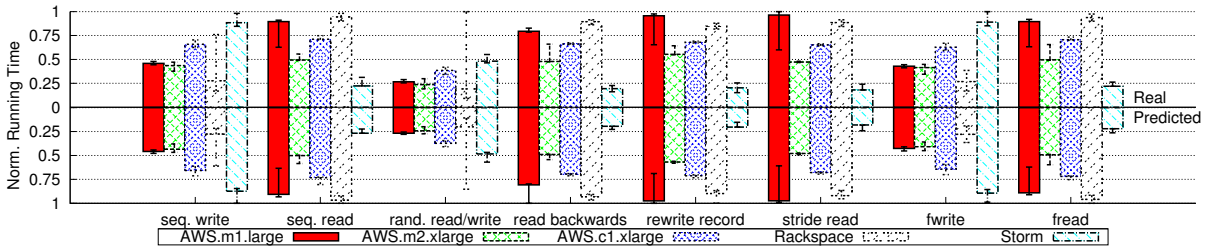


Figure 5: **The real IOzone running time and the predicted running time on five different instances. X-axis shows the I/O pattern, and y-axis shows the normalized running time.**

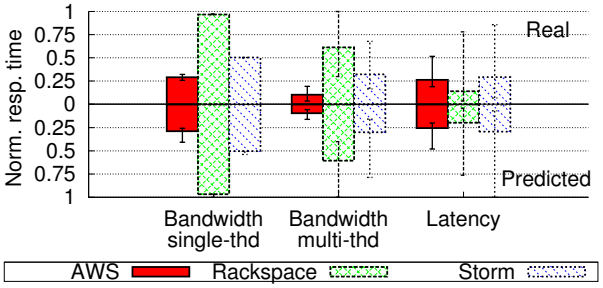
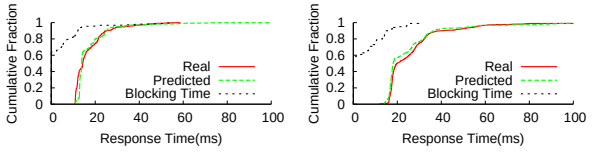


Figure 7: **The real and predicted downloading time for both bandwidth-bound and latency-bound cases. For bandwidth-bound case, we further test two scenarios with single and multiple downloading threads. X-axis shows the test cases, and y-axis shows the normalized downloading time.**



(a) AWS c1.xlarge (b) Rackspace

Figure 8: **The real and predicted response time distribution for the custom lock benchmark application using log4j. The blocking time of each request during the replay run is also shown. Here we show the results on two instances: AWS' c1.xlarge and Rackspace, while the ones on other instances are similar.**

databases (RDS and Xeround), and omit the ones with unmanaged database instances. As can be seen, in both single- and multi-thread cases, the prediction accuracy is high. This is because CloudProphet faithfully replays all MySQL queries.

5.2.3 Network Prediction

Figure 7 shows the real and predicted downloading time of the file serving website under both bandwidth- and latency-bound cases. We set up the website in AWS, Rackspace, and Storm's default data centers. For the bandwidth-bound case, a 1GB file is downloaded by a client inside the data center; for the latency-bound case, a 10KB file is downloaded by multiple clients distributed over 200 PlanetLab [17] nodes. For the bandwidth-bound case, we further test a multi-threaded scenario with 10 clients downloading simultaneously. As can be seen, CloudProphet accurately predicts the downloading time under both bandwidth- and latency-bound cases. This is because the cloud replayer faithfully issues the same network library calls as the original application.

5.2.4 Lock Prediction

Figure 8 shows the real and predicted response time distribution for our custom web application using log4j. It also shows the blocking time of each request during the replay run to illustrate lock contention. In the application, each request writes 2MB of log in a critical session protected by Java monitor. We send 30 requests per second to trigger lock contention. From the figure, lock contention occurs for around 40% of the requests, because the blocking time (and the response time) starts to increase at the 60th percentile point. CloudProphet can accurately predict the contention effect, as the two distributions match well with each other. Further, log4j uses Java monitor enter/exit to protect the critical session, while CloudProphet replays the lock logic with simple pthread locks. This suggests that the exact lock implementation is unimportant for performance prediction, as long as the lock logic is faithfully replayed.

5.3 Real Application Study

In this section, we evaluate how well CloudProphet can predict the end-to-end response time of real multi-tiered web applications. Specifically, we seek to answer the following questions:

- Can CloudProphet accurately predict the end-to-end response time of real web applications?
- Is the resource usage to serve a request similar between on-premise and in cloud?
- Can CloudProphet predict multiple resources simultaneously?
- How much can dependency extraction help?
- How well does CloudProphet cope with different load levels?

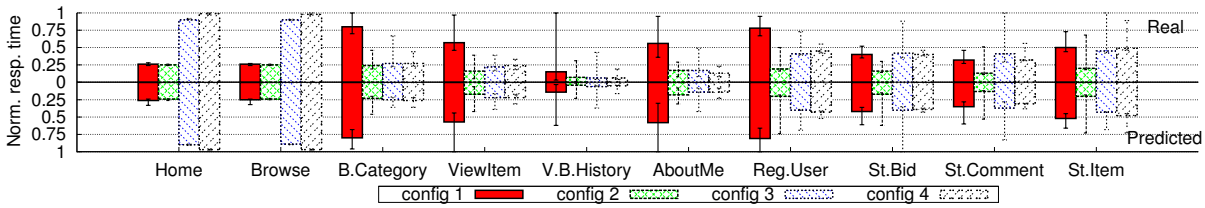


Figure 9: **The normalized response time of 10 RUBiS request types, shown along the x-axis, on four cloud migration configurations. The real and predicted time is shown in the top and bottom half of the figure. Time is normalized per cluster.**

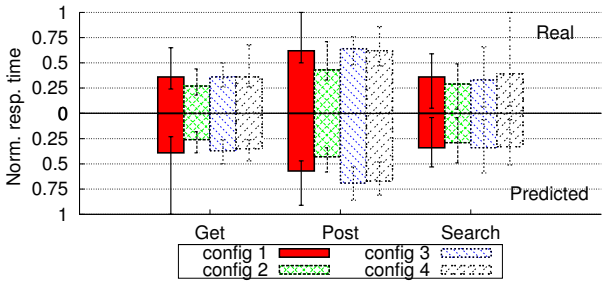


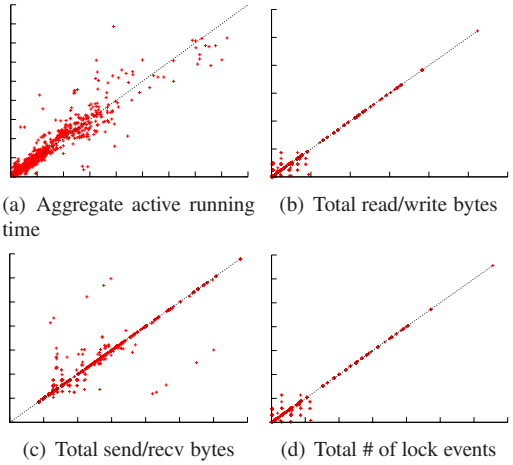
Figure 10: **The normalized response time of 3 MediaWiki request types on the four cloud configurations.**

5.3.1 End-to-end Prediction Accuracy under Normal Load

We first evaluate CloudProphet's end-to-end prediction accuracy under normal load level. We configure the RUBiS client to send requests using 20 concurrent user sessions and the MediaWiki client to send request at a rate of 5 requests per second. Under such load level, the two applications are not heavily loaded both on-premise and in cloud, with the average CPU load below 50%. We use this setting for most results in the next few sections.

Figure 9 and 10 show the overall end-to-end prediction results of the two applications under normal load. Each cluster of bars corresponds to one request type determined by the web page visited. The results show that CloudProphet can predict the end-to-end response time accurately in most cases. The relative prediction error of all request types in both applications is below 10%, except for V.B.History on config 2, which has a relative error of 24%. In the figure, we only show 10 out of the 26 request types for RUBiS. Other types have similar results. For MediaWiki, we show all three request types (Get, Post and Search).

Three key factors are critical to CloudProphet's accuracy. First, the resource usage required to handle a request is similar between on-premise and in cloud. If the resource usages are significantly different, the traces collected on-premise cannot be used to predict the usage in cloud. Second, CloudProphet can cover multiple resource types. As we will see later, the two applications, especially MediaWiki, use multiple types of resource simultaneously, and CloudProphet is able to predict the time spent on each resource type accurately. Finally, CloudProphet can precisely extract the application dependency, so that the prediction results are not affected by any false dependency. In the following sections, we evaluate each one of them in turn.



(a) Aggregate active running time (b) Total read/write bytes

(c) Total send/recv bytes (d) Total # of lock events

Figure 11: **The resource usage similarity of RUBiS requests between on-premise and in cloud (config 4). Four dimensions are shown. Each point corresponds to a request, with its X/Y coordinate being the value observed on-premise/in-cloud.**

5.3.2 Resource Usage Similarity

Figure 11 shows how similar the resource usage is between on-premise and in cloud (config 4) for RUBiS requests. We compare the resource usage to serve each request along four performance-critical dimensions: the aggregate active running time, the total read/written bytes, the total send/recv bytes, and the total number of lock events. We also compare along other dimensions, such as the number of MySQL queries, as well as all MediaWiki's requests, and the results are similar. Each point in a plot corresponds to a request, and its X/Y coordinates correspond to the values observed on-premise/in cloud. As can be seen, most of the points aggregate close to the $y = x$ line, which suggests that the requests have similar resource usage both on-premise and in cloud. Note that the results for total read/written bytes and for total number of lock events are similar, because in RUBiS disk I/O operations are always protected by locks.

There are also a few interesting outliers. In the total

send/recv bytes plot, there are about 10 points located far away from the center line. Investigation shows that they are caused by the initialization overhead of the first request sent over a connection between the web and the application server. Because the requests' arrival order may change, the first request of a connection can also change, which leads to the non-determinism. Such incidents are rare. Given that there are about 2000 points in a plot, it only happens to 10/2000=0.5% of all requests.

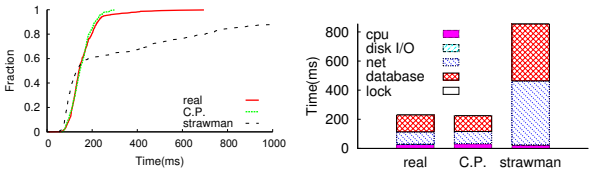
5.3.3 Predicting Multiple Resources

Figure 12(a) shows the median time spent on each resource type for three request types in RUBiS. The time is broken down along the processing path of each request. Since there is no parallel processing in each request, the time spent on different resource types does not overlap, and their sum equals to the total end-to-end response time. As can be seen, the dominant factor of the end-to-end response time changes drastically across different cloud configurations. For instance, St.Item's dominant factor is database in config 1, but it shifts to network in config 3 and 4. It is difficult to find a representative benchmark for such workload, or to decide when to use a network-intensive benchmark and when to use a database-intensive one. In contrast, because CloudProphet can predict the time spent on both resources accurately, it can faithfully reproduce the shifting of the dominant factor and produce accurate end-to-end prediction results.

Figure 12(b) shows the time broken-down results for MediaWiki. As can be seen, MediaWiki has more diverse resource usage than RUBiS. It has non-trivial usage for all resource types except for lock. This further complicates the prediction using benchmarks. For instance, request type Search, given comparable time spent on each resource, has no single dominant factor in the end-to-end response time. If one wants to use benchmarks to predict the performance, she needs to understand how the total time is composed by the time spent on each resource, and selects the benchmark with the same characteristics. Even this can be plausibly done with a profiler, the time distribution among different resources can change across different cloud configurations. On the contrary, because CloudProphet can trace and replay the multiple types of resource usage with high fidelity, it can accurately predict the end-to-end response time.

5.3.4 Benefit of Dependency Extraction

In this section, we evaluate how dependency extraction (§ 3.2) helps the prediction accuracy. To do so, we implement a strawman prediction tool without dependency extraction. The tool simply replays the events in each thread's trace sequentially as shown in § 2. We then compare the prediction accuracy of the strawman with that of CloudProphet.



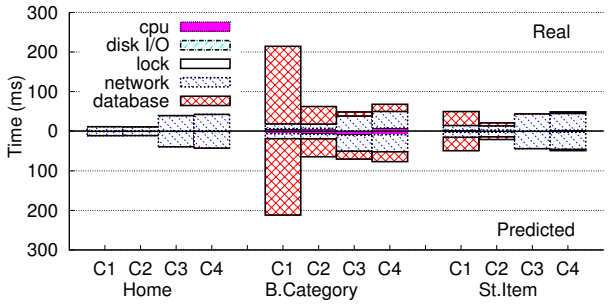
(a) Time distribution (b) Time broken-down

Figure 13: **(a) 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. (b) The time broken-down by resource types for the 85th-percentile request.**

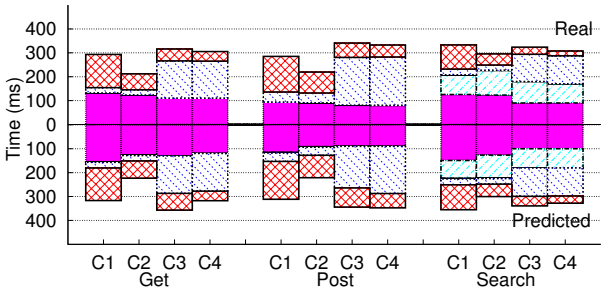
Figure 13(a) shows the response time distribution of RUBiS' AboutMe request under real run, CloudProphet's prediction, and the strawman's prediction. The strawman tool over-predicts the response time of a portion of the requests by many folds. Figure 13(b) shows the time broken-down of a request at the 85th-percentile. As can be seen, both the network and database time increase significantly under the strawman tool. We find that the increment of network time is due to false dependency, as shown in § 2. In one scenario, two internal requests received by the same application server thread are actually sent by two different web server threads. During replay, because the cloud instance has a different CPU type from the on-premise one, the two web server threads are scheduled differently, which causes the requests to arrive out-of-order at the application server. Therefore, the first arrived request will be blocked and not received by the application server's replayer until the second one has arrived, introducing additional network time.

It is interesting to see that the database time is also increased significantly. We find this is due to the increased burstiness of the MySQL queries. In some cases, the out-of-order has become serious that many incoming requests are blocked by the same request at the application server. Once the culprit request has arrived, the other requests are unblocked and processed in a tight batch. Therefore, the database server can be heavily loaded when multiple application server threads are unblocked at similar time. This suggests that the false dependency can also introduce additional prediction error because of the changed resource usage pattern.

On the other hand, the predicted response time distribution by CloudProphet matches very well with that of the real run, because CloudProphet can extract the true dependency between a request and an event block. By enforcing the dependency, CloudProphet faithfully reproduces the same resource usage pattern as the real application. The results in this section are from config 2, and we also observe similar phenomena on other configurations.

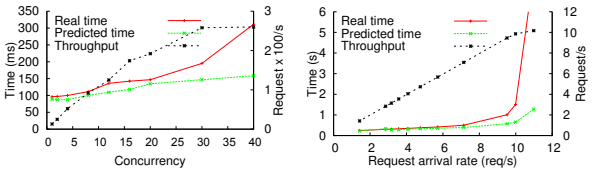


(a) RUBiS



(b) MediaWiki

Figure 12: **The response time broken down by resource types on four cloud migration configurations (C*n* = config *n*). For each resource type, the median time is shown.**



(a) RUBiS AboutMe (b) MediaWiki Get

Figure 14: **The median real and predicted response time at different load levels, together with the corresponding real throughput.**

App	Response time (ms) va.	w/t	w/st	Throughput (req/s) va.	w/t	w/st
RUBiS	79.1	84.4	87.4	257	257	257
MediaWiki	276	290	309	35.7	31.7	28.6

Table 4: **The response time and the throughput of the applications with (w/t) and without tracing (va.). We also show the results with a less optimized tracing engine using the standard memory management scheme (the w/st columns).**

5.3.5 Prediction Accuracy at Different Load Levels

The previous sections show that CloudProphet can achieve high prediction accuracy under normal load. In this section, we evaluate how the prediction accuracy changes under different load levels. It is not the design goal of CloudProphet to predict the response time under heavy load, because a healthy web application should stay in its normal load range during most of the time to offer low-latency and predictable service. Further, accurately predicting the response time under load is inherently difficult. When an application is overloaded, its response time become very sensitive to the system load, and even some minor error in resource usage replay can change the system load, and result in significant deviation in response time. The goal of this experiment is to understand where the limit is.

Figure 14 shows the median real and predicted response time of two request types at different load levels. We choose the two request types because each of them is among the heaviest request types in its corresponding application. The experiment of the two request types run on config 4 and config 1 respectively. We choose the two configurations because they are less powerful than our on-premise deployment. We can then collect the event traces under normal load and replay at heavy load. In each plot, we also show the overall throughput of the application to indicate when it has achieved its maximum throughput (*i.e.*, bottlenecked). As can be seen, the predicted time matches the real one for a wide range of load levels before the application is bottlenecked. For the AboutMe request, the predicted time matches the real time well until the bottleneck point, where the gap starts to grow. For the Get request, the gap starts to grow significantly when the application has reached 70% of its maximum throughput. When the load is high, CloudProphet under-predicts the response time in both cases, because the application might be bottlenecked by a resource not captured by CloudProphet, such as memory bandwidth.

5.4 Overhead

In this section, we evaluate the computation and storage overhead of CloudProphet while collecting the trace, and compare its deployment overhead with the migration overhead of the real application.

Computation overhead We evaluate the tracing engine's computation overhead by measuring its impact to both the response time and the throughput of the on-premise applications. For response time, we show the median time of the request type that is most affected by the tracing overhead. Table 4 summarizes the results. We can see the tracing overhead is low for both the response time and throughput. The relative response time increment and throughput decrement is less than 12%. Note that the overhead is not carried over to the prediction error, because our tracing engine excludes the overhead from the event traces. For RUBiS, the tracing overhead on throughput is negligible, possibly because RUBiS is not CPU intensive.

We also compare the overhead with a less-optimized tracing engine using the standard memory management scheme (§ 4). The results show that the custom memory management scheme used in CloudProphet can almost

halve the overhead in some cases. It is especially effective with MediaWiki, where the event density is high and the workload is CPU-intensive.

Storage overhead We measure the trace size of each application, and compute the storage overhead per request. The results show the overhead of MediaWiki is 60.2KB/req, while RUBiS' is 16.5KB/req. At MediaWiki's overhead level, an one hour trace of a busy application that serves 10K requests per minute [44] is less than 4GB. Such storage requirement is manageable even with regular PCs. Further, the incoming bandwidth of a cloud is usually provided free-of-charge [18, 25] to incentivize people to move their assets into the cloud. Therefore, CloudProphet does not incur additional costs to upload the event traces to cloud.

Deployment overhead Table 5 shows a qualitative comparison between the overhead of deploying CloudProphet and migrating the real application. CloudProphet can cut the overhead in many aspects. First, it does not require the installation of any dependent software, unlike the real applications. During our experiments, we find that installing all the dependency can be a headache, simply because there are too many dependency requirements to satisfy. Second, CloudProphet does not require any software license, because it does not contain any real application logic. Third, migrating an application requires moving all of the application data files, including configurations, user data, libraries, etc. In contrast, CloudProphet only requires uploading the event traces for replay. Finally, both deploying CloudProphet and real application requires migrating the database to the cloud. We consider this as an acceptable overhead as database migration is a common practice and can be easily automated.

6 Related Work

To the best of our knowledge, CloudProphet is the first system that can accurately predict the end-to-end performance of real applications in cloud. The problem of performance prediction has also been considered in many other contexts.

Hoste *et al.* [34] propose to use architecture-independent characteristics to find the most similar benchmarks to predict the performance of CPU-intensive applications across a large collection of CPU types. The characteristics are collected through heavy instrumentation. In contrast, CloudProphet only captures the active running time with lightweight technique, and uses simple scaling to predict the running time in cloud. Our evaluation shows the prediction accuracy is high on a variety of cloud instances.

//Trace [41] predicts the performance of an I/O intensive application across different storage systems through I/O trace replay. It also uses a throttling technique to extract data dependency between different application threads. CloudProphet also traces and replays disk storage events in a similar way as //Trace, but targets at a different dependency model specific to web applications.

Much work has been done to predict web application's performance through modeling [42, 45]. The work mostly focuses on predicting the performance across different load levels or different component placement strategies. CloudProphet does not require modeling either the application or the platform.

WebProphet [38] is a system to predict the page load time of a web service under different deployment scenarios. It uses a throttling technique to extract the dependency between different web objects, and then simulates the page loading process given the deployment configuration. CloudProphet focuses on predicting the end-to-end response time of each web request. It uses emulation instead of simulation to test the cloud platforms by consuming real resource.

7 Conclusion

A fast growing business, public cloud computing has already become mainstream. However, the diverse hardware and virtualization technologies have led to dramatically different cloud performance, and it has become ever-difficult to reason about the application performance inside cloud platforms. In such context, we present CloudProphet, the first end-to-end performance prediction tool for web applications in cloud. We identify and address two key challenges, including how to capture the application's resource usage with the right level of detail, and how to extract and enforce the application dependency. Despite having some limitations, CloudProphet can accurately predict the end-to-end performance of two real applications over a variety of cloud platforms, while achieving low tracing and deployment overhead. We believe CloudProphet represents a significant first step in enabling fast and performance-informed cloud platform selection.

References

[1] Amazon Relational Database Service. http://aws.amazon.com/rds/.

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

[3] Amazons physical hardware and EC2 compute unit. http://huanliu.wordpress.com/2010/06/14/amazons-physical-hardware-and-ec2-compute-unit/.

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

[5] clock_gettime man page. http://www.kernel.org/doc/man-pages/online/pages/man2/clock_gettime.2.html".

[6] CloudHarmony. http://cloudharmony.com/benchmarks.

[7] Comparision between Amazon web services (AWS) or Rackspace cloud servers? http://stackoverflow.com/questions/6397587/comparision-between-amazon-web-services-aws-or-rackspace-cloud-servers.

[8] Dell Announces Its First Public Cloud Offering. http://content.dell.com/us/en/corp/d/press-releases/2011-8-29-dell-vmware-public-cloud-datacenter.aspx.

[9] GetThreadTimes Function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms683237(v=vs.85).aspx.

	Installation	Software licensing	File migration	DB migration
App migration	High (many dependencies)	May require	High (all data files)	Required
CloudProphet	Low (the replayers only)	Not required	Low (event traces only)	Required

Table 5: **A qualitative comparison between CloudProphet's deployment overhead and the migration overhead of the real application along various dimensions.**

[10] HP Enterprise Cloud Services. http://www.hp.com/enterprise/cloud.

[11] IOzone Filesystem Benchmark. http://www.iozone.org.

[12] log4j Logging Service. http://logging.apache.org/log4j/1.2.

[13] MediaWiki. http://www.mediawiki.org.

[14] MediaWiki Lucene Search Extension. http://www.mediawiki.org/wiki/Extension:Lucene-search.

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

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

[17] PlanetLab. http://www.planet-lab.org.

[18] Rackspace Cloud. http://www.rackspacecloud.com.

[19] Rackspace Cloud Servers versus Amazon EC2: Performance Analysis. http://www.thebitsource.com/featured-posts/rackspace-cloud-servers-versus-amazon-ec2-performance-analysis/.

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

[21] SPEC CPU2006 Benchmark. http://www.spec.org/cpu2006.

[22] Storm, a Liquid Web Company. https://www.stormondemand.com.

[23] Tcpreplay. http://tcpreplay.synfin.net.

[24] TPC-C: On-line Transaction Processing Benchmark. http://www.tpc.org/tpcc.

[25] Web Hosting in Amazon AWS. http://aws.amazon.com/web-hosting.

[26] Xeround Cloud Database. http://www.xeround.com/.

[27] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03*, pages 74–89. ACM, 2003.

[28] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *USENIX FAST'04*, pages 45–58. USENIX Association, 2004.

[29] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A file system to trace them all. In *USENIX FAST'04*, pages 129–145. USENIX Association, 2004.

[30] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl. Automating network application dependency discovery: experiences, limitations, and new solutions. In *OSDI'08*, pages 117–130. USENIX Association, 2008.

[31] DeterLab. http://www.deterlab.net.

[32] W.-c. Feng, A. Goel, A. Bezzaz, W.-c. Feng, and J. Walpole. Tcpivo: a high-performance packet replay engine. In *MoMeTools*, 2003.

[33] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. In *NSDI'07*, pages 20–20. USENIX Association, 2007.

[34] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere. Performance prediction based on inherent program similarity. In *In PACT*, pages 114–122. ACM Press, 2006.

[35] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan. Request extraction in magpie: events, schemas and temporal joins. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM, 2004.

[36] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *USENIX FAST'05*, pages 25–25. USENIX Association, 2005.

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

[38] 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.

[39] 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.

[40] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox, 2008.

[41] 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.

[42] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI'05*, pages 71–84. USENIX Association, 2005.

[43] 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.

[44] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[45] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *In Proc. of the ACM SIGMETRICS2005*, pages 291–302, 2005.

[46] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *IEEE INFOCOM*, 2010.