Estimating Application Behavior in New Environments

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

Abstract – Networked applications often need to change their running environments for many reasons such as migrating to the cloud or coping with new application demands. However, there lacks an efficient and accurate approach to estimate how an application behaves in a new environment, posing a challenge for application owners to decide what the most appropriate environments for their applications. This paper describes CloudProphet, a new approach to estimate an application's behavior without porting it to many available environments and without abstracting away its complex behavior. CloudProphet uses easy-to-generate shadow programs to mimic an application's resource usage in a new environment, and run the shadows in a target environment to estimate the application's performance. Our evaluation with two popular cloud platforms shows that CloudProphet can help customers identify the best-performing cloud environment and accurately estimate the performance and cost for a variety of applications.

1 Introduction

The environments in which networked applications operate change. Beyond the traditional drivers of change such as new hardware and software, the availability of affordable, scalable, and geo-distributed resources [6, 19, 23] is luring applications to the cloud [1]. Further, as an application or its workload evolves, a different environment may become more suitable. Facebook reports re-architecting its photo store, roughly once every few months, due to changing demands for memory and network as well as increasing request rates [30].

In these contexts, it is useful to assess an application's performance in different environments. First, a good estimate of how an application performs in a new environment allows an application owner to choose the most appropriate environment for the application among the many possibilities that differ vastly in cost and performance. For example, the I/O bandwidth offered by one cloud can be twice as high as that of a similarly priced instance from another cloud [24] and no cloud excels across all resource types, i.e., computation, storage, and network performance [38]. Questions like "should I go with Amazon AWS or Microsoft Azure?", and "is the large (xlarge) instance sufficient to handle my workload?" are frequent topics in public forums and mailing lists [12]. Second, a quick estimate can speed up the process to re-architect a system when application workloads change, which is a competitive advantage.

A straightforward approach to assess an application's performance is to port an application to each new environment. This approach can accurately assess an application's behavior, but requires onerous manual efforts and is challenging in practice. This is because applications can be tightly coupled with their environments. For example, many legacy applications deployed inside on-premise datacenters are not packaged into portable virtual machine images, and tasks such as installing the required software and migrating the application configurations can incur significant overhead [47, 51].

Another common approach to predict application behavior is modeling. A plethora of work [42–44, 48] has emerged to estimate the application performance under different environments and application parameters, such as component placement [44], physical CPUs [43], workload mix [42], and the number of replications [44, 48]. Though the modeling approach allows efficient exploration of the parameter space without onerous application porting effort, it requires tremendous expert effort to build and calibrate these models [52], while each of them only captures one or few aspects of the environment and application. A holistic model that can accurately capture all the dimensions of a complex networked application running in a completely new environment (e.g., a cloud platform) is still missing, likely because of the sheer complexity to describe everything analytically.

In this work, we aim to achieve the best of both worlds by exploring a middle approach. We ask the question: can we estimate the performance of a networked application in a new environment without full application porting and yet without abstracting away the essential details of the application and its environment? Our answer is both yes and no. We discover that for generic applications, behavior prediction is extremely challenging (if not intractable), as an application can behave as an adversary to defeat any prediction method. For instance, an application could change its behavior after detecting a new environment. In this case, one cannot predict an application's behavior without actually running it in a new environment.

Despite the negative answer for generic applications, we find that we can predict the performance of a broad range of real applications in different environments with-

[1]This work was done when Ang Li and Xuanran Zong were at Duke University

out porting or analytically modeling them, as long as these applications do not vary their application logic when environments change and their inter-component dependencies can be extracted.

We build CloudProphet which estimates an application's behavior by running a set of high-fidelity *shadows* in a target environment, in place of the real application components. The shadows mimic the *resource usage* of their corresponding components with easy-to-port code agnostic to the application logic. For instance, a shadow will read or write the same number of bytes through disk or network I/O as its corresponding component. Also, CloudProphet enforces the same inter-component dependencies among the shadows as the real application does, *e.g.*, a shadow for a front-end server will contact the back-end shadow in the same way as the real front-end server does. CloudProphet builds the shadows automatically by tracing the resource usage of the application components, and the shadows simply replay the traces to reproduce the same usage.

Our evaluation shows that mimic'ing just the resource usage and the dependencies suffices to estimate a variety of practical applications' behavior in a new environment with high fidelity. This result holds for the applications and cloud platforms we tested. The cloud platforms we tested differ from each other along many dimensions, including CPU types, network conditions, and interference levels. This is our primary contribution. With CloudProphet, we can estimate the application performance seen by users, the scalability of the application, and the expected running cost in heterogeneous environments.

There are a couple reasons why this approach works. First, at the level of individual components, we find that the performance and monetary cost of running a component in a new environment depend on the resource usage. For instance, cloud billing policies bill based on the usage of a few key resources [38]. Similarly, the performance is a function of the infrastructure (hardware and software) and the resource competition from others on the shared infrastructure. Both of these can be exercised by replaying the actual resource usage. Second, at the level of an entire networked application, the inter-component dependencies are critical to the end-to-end performance, and they can be extracted and enforced precisely.

We evaluate CloudProphet in the context of cloud migration with two popular public cloud providers: Amazon AWS [6] and Joyent [15]. Our results show that CloudProphet can help customers identify the best-performing cloud platform and instance type for two multi-tiered web applications, RUBiS [25] and MediaWiki [16], while following benchmarking and modeling results would result in suboptimal choices with only half the optimal performance. Besides performance, CloudProphet can also help customers estimate the cost of running a migrated application. Furthermore, we find CloudProphet achieves high prediction accuracy over a number of applications with diverse resource demands. Finally, we find CloudProphet imposes low tracing and replay overhead, and deploying CloudProphet is much easier than porting the real applications.

2 Overview

In this section, we describe a few use cases for CloudProphet. Then, we sketch a solution idea and discuss the challenges we faced and the design choices we made. Further details are deferred to §3.

2.1 Use Cases

We provide several use cases in which three conditions hold. First, there is a need to transition applications to a different environment from where they currently operate. Second, making the transition would require the knowledge of the application behavior in the new environments – performance, cost, and scalability. Third, porting the applications to the new environments is difficult and costly. Together, these conditions mean that there is value in predicting the application behavior in the new environments.

Migrating Applications to the Cloud: There is an increasing desire to migrate enterprise applications to the public cloud infrastructures [1]. The cloud providers accrue savings from the economies of scale. Customers pay only for what they use, and avoid the expense of operating their own infrastructure, and can potentially scale better [29].

A key challenge in migrating applications to the cloud is to choose the cloud environments best suited for the applications. On the one hand, IaaS (Infrastructure-as-a-Service) providers allow migrating binaries but there is much variation in performance vs. price across providers – 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 [24]. Further, no single cloud excels across all resource types– computation, storage, and network [38]; for example, a cloud that offers faster virtual instances has lower inter-DC bandwidth [38]. Hence, the best-suited cloud for an application depends on its resource demands, which are not readily quantifiable. On the other hand, even within each provider there are many tiers of service. For instance, Amazon AWS offers 13 instance types, ranging from small– one shared core and 613MB memory to the very large– 16 dedicated cores and 60.5GB memory. The smaller instances are cheaper but have more interference from co-located applications [49] whereas the resources offered by the larger instances may not justify their extra costs if the application is not bottlenecked at those resources.

A naive approach would be to port the application to each candidate environment and measure its behavior. However, the approach can be difficult and unattractive for a few reasons. First, an application usually depends on many third-party software components, and installing these components in a new environment is non-trivial. A simple web application may depend on many components such as web servers, script interpreters, and various caching systems; a distributed application may depend on data serialization libraries such as Google protocol buffers [21] and Apache thrift [8]. These components are available in diverse forms (binary packages or source code), which makes the installation process difficult to automate. Moreover, some licensed components might require additional authorization to run in the cloud.

Second, porting the application configurations can also be difficult. The configurations of a large distributed application are notoriously difficult to manage even in a single environment [33], because different components are usually built in ad-hoc ways and there lacks a standard way to describe their behavior. Porting the configurations to a new environment can be even trickier, because one needs to figure out which portion of the configurations needs to be updated. Recent configuration management tools such as Chef [10] and Puppet [22] offer a high-level environment-independent language to describe the component behavior and automatically generate the low-level configurations. However, they can help only when the configurations are already under their management, while many legacy applications still run with hand-crafted configuration files.

Geo-distribution: A more recent use case is for enterprises to place some application components in the cloud. This lets enterprises make use of the geographical diversity of the cloud to reduce the latency to the customers. By adding redundancy, this can also keep the applications more available.

The challenges to pick the best-suited cloud environment – the abundance of choices and that the potential gains depend on application type – remain. Additionally, one has to choose how to split the application between the enterprise and the cloud. Splitting apart a chatty pair of components that need high bandwidth (or low latency) is risky. Also, one needs to account for the wide area network characteristics, including both performance and cost, offered by the various providers.

Changing workload: Changes in user request rate or workload mix force frequent re-design. Facebook reports that its photo storage service changes design about once every six months [30]. While some of these changes are simple – adding more servers to a tier, others can involve very complex ramifications – e.g., increasing consistency using write-through instead of write-back. Instead of relying on the *qualitative* knowledge of the human experts to explore the design space, what if we had a mechanism to *quantitatively* predict the application behavior under a new workload?

2.2 Solution Idea

How can we predict the performance of an application in a new environment without going through the pain of porting it? The basic idea is rather simple. Replace each component of the application with a shadow that mimics the component's behavior, i.e., takes the same inputs, generates the same outputs, and makes the same demands on all resources (computation, memory, storage, network, ...), but is made up of very simple code and hence is trivial to port. If we can create these shadows, predicting behavior is easier – deploy the shadow in potential environments (public clouds, server SKUs, configurations) and compute the observed performance, cost, and scaling behavior.

We propose to trace the behavior of each component, i.e., its inputs, outputs, and resource usage. The shadow then replays the collected trace while exerting the same demands on every resource (CPU, network, ...). CloudProphet traces at system call boundaries and tags each trace event by the thread that generates it. System call parameters and return values let us capture inputs and outputs. Some resource use is implicitly captured in the call parameters, e.g., network transfers. CloudProphet explicitly captures the usage of resources between every pair of successive trace events – e.g., the number of CPU cycles used.

The approach is based on two key assumptions. First, the application is not *adaptive* to the environment where it operates. An application is adaptive if it can change its resource demands based on certain environment characteristics. For instance, an adaptive network transfer application might first probe the available bandwidth in its environment and decide how many TCP flows to use. It is very difficult to predict the behavior of adaptive applications, because they can enter a new untraced code path when running in the new environment.

Second, the target application's dependency must be available to correctly reproduce the order of the events on multiple components. Otherwise, the replay can deviate significantly from the original application's behavior, causing prediction error [39]. Dependency extraction is an old and challenging problem: many tools and techniques have been proposed to extract dependency from various application types with different accuracy levels [31, 32, 35, 37, 39, 41]. In this paper, we propose a new technique to transparently extract and enforce the dependency from applications that follow the dispatcher-worker programming pattern, which is popular in many

Domain	Servers
Web	Apache, IIS, Tomcat
RPC	Java RMI, ASP.NET
Middleware	JBoss, PHP fastCGI, JOnAS, WebSphere
SCM	CVS, Subversion, Git
Authentication	Kerberos, OpenLDAP

Table 1: **Popular servers in different domains that follow the dispatcher-worker pattern.**

domains (Table 1). The technique complements the existing work, and our evaluation shows it allows CloudProphet to accurately predict the behavior of a variety of applications. On the other hand, the deployment of pervasive tracing tools such as X-trace [32] and Dapper [41] can also benefit CloudProphet as these tools can also produce accurate dependency graphs.

2.3 Challenges

Resource-Mimic'ing Replay: CloudProphet sets up a shadow for each component (application thread) and replays the trace events such that between each event pair the shadow uses the same resources as the original.

A primary challenge is to translate the resources used in the observed environment to those that will be used in the new environment. This is easy for some resources– network transfer sizes are likely to remain the same, but not for others– how to translate a CPU use of x cycles (or y seconds) to the different types of CPUs in the new environments. We will show in §3.1 how CloudProphet mimics each of the various resource types.

A subsidiary challenge is the choice of resources to mimic since there is a trade-off between fidelity and overhead. CloudProphet mimics all the major resource types–computation, storage, network and some IPC such as locks. Our results show that these resource types are necessary– there are applications and environments that can bottleneck at each of these resources, and hence, not mimic'ing them leads to poor predictions. We note also that CloudProphet does not trace some other aspects (e.g., cache misses, individual memory accesses). Details are in §3.1, but we note here that CloudProphet employs a work around for each traced resource that achieves, for realistic applications, a fidelity close to the brute-force approach of tracing everything.

Transparent Dependency Extraction: Though in this paper we only focus on dispatcher-worker applications, it is still non-trivial to extract their dependency without any intrusive instrumentation. The key challenge is that the requests can be scheduled by the dispatcher to be served by different workers. Consider the example in Figure 1. In the trace (see left), it happens that requests A and B are served by the leftmost thread. But, in the new environment, request A may take longer to process causing request B to be served by a different thread (see right).

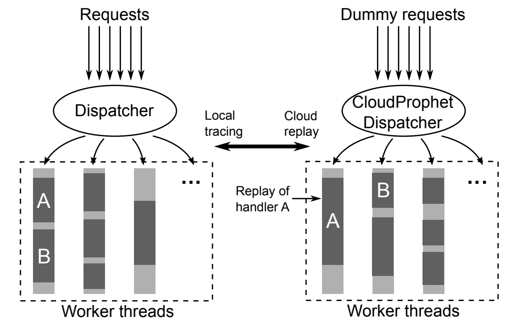


Figure 1: **This figure shows how CloudProphet replays the behavior of each application component leveraging the dispatcher-worker pattern. CloudProphet uses its own dispatcher to mimic the real dispatcher's behavior, and replays the events in each request's handler.**

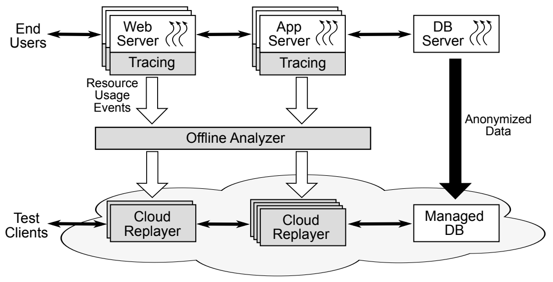


Figure 2: **The architecture of CloudProphet as applied to a three-tier web app. The key parts of CloudProphet are shown with shaded boxes: the tracer, an offline analyzer, and the replayers (shadows).**

This can cause problem if we naively replay the events in each thread according to their traced order.

Details on how CloudProphet accurately extracts the dependency are in §3.2.2. In short, CloudProphet leverages the pattern and uses only the network events to extract the real application dependency. During replay, CloudProphet uses a special shadow to mimic the functionality of the dispatcher, i.e., assign incoming requests to the idle shadow workers.

3 Design Details

Here, we describe how CloudProphet traces and replays to mimic the resource usage of the actual application but with much simpler shadows. We also describe how dependencies between the components are enforced.

3.1 Resource Mimic'ing Replay

Let's examine this one major resource at a time.

Computation: How to faithfully replay the computation done between two trace events? There are many potential characteristics– instruction mix, memory access pattern, and instruction-level parallelism– that affect the runtime in the new environment. Collecting these requires instruction-level instrumentation which is expensive. For instance, just tracing the instruction-level paral-

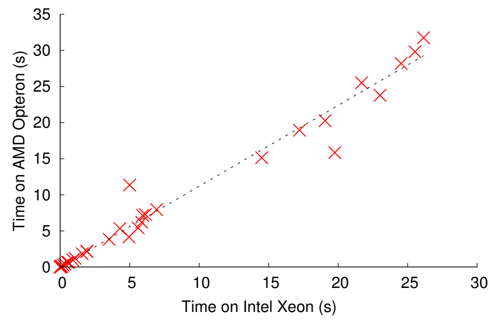


Figure 3: **Comparing the runtime of SPEC CPU2006 benchmarks on two processors (AMD Opteron 2374HE vs. Intel Xeon E5507). There is one point per benchmark. The dashed line shows the best linear fit.**

lelism slows down an app by over 600X [17]. Reproducing these characteristics with simple code is also hard.

To understand the problem better, we ran the SPEC CPU2006 benchmarks on nine server processors, including those used at three popular public cloud (AWS, Rackspace, and Joyent). These CPUs vary in frequencies, vendors, architectures (Intel's Core and Nehalem, AMD's Shanghai), cache configurations, and the year of manufacture ranges from 2007 to 2010.

Figure 3 shows the results for a processor pair. It compares the runtime of the SPEC benchmarks between Intel's Xeon E5507 (Nehalem) and AMD's Opteron 2374HE (Shanghai). But for two outliers, the runtime of the various benchmarks conforms to a linear fit. This observation holds for all the processor pairs tested– the smallest pearson correlation coefficient was 0.95.

But why should such a linear fit exist? We have two conjectures. First, unlike highly optimized code, general purpose code tends to rely less on the esoterics of processor architecture. Second, that competition has lead to significant uniformity across architectures.

CloudProphet exploits this linear fit. Given the runtime on one processor, it predicts the runtime on another processor by (linearly) scaling with the appropriate factor. The shadow runs in a busy loop for this duration. We note that while we could create adversarial workload (low cache hit rate and large working set) on which this simple approach yields poor predictions, we are surprised that the approach works for real apps (see §5).

Storage: CloudProphet mimic's the use of two types of storage – raw storage, by which we mean accesses to files and blob, and managed database services offered by popular clouds, such as Amazon's Relational Database Service (RDS) [3] and Microsoft's SQL Azure [18]. We ignore a few others that are less commonly used by enterprise apps (key-value [5] and queue storage [4]).

Raw storage is relatively easy. By intercepting relevant system calls (e.g., `read, write, fsync`), CloudProphet records the amount of data read or written and the context such as the inode and the position of the file pointer. The actual content is silently dropped. Shadows mimic the calls but with fake content. CloudProphet performs bookkeeping when necessary, e.g., opening files.

As before, there is adversarial workload that circumvents this approach, e.g., operations on memory mapped files. We do not know how to trace such operations cheaply; raising a page fault upon every access to a mapped page requires kernel changes and incurs high tracing overhead. In our evaluated applications (§5), we find that memory-mapped files are small and fit in memory. Therefore, missing those I/O operations did not introduce significant error.

Managed Databases: It is a trend for cloud providers to offer managed database services since the regular VMs on offer cannot satisfy the high availability requirements of databases. To resource mimic the accesses to these databases, CloudProphet special-cases all database related system calls during tracing. Prior to replay, CloudProphet migrates the application data to the chosen database service. During replay, the shadows make the same queries. For data privacy, we note that the actual content in the database does not matter and can be anonymized as long as the schemas and the data properties, such as the distribution of key distributions, that impact performance are unchanged.

Network: By tracing network system calls, CloudProphet records the size and the four-tuple (src_addr, src_port, $dest\_addr$, $dest\_port$) identifying the connection for each network transfer. We show later how this helps to identify dispatcher and worker threads (§3.2). CloudProphet also records socket-level parameters that may be set by the application (e.g., via `setsockopt` calls) since these options can impact network performance considerably [50]. Shadows replay the same send/recv/setsockopt calls.

Due to the state maintained in TCP and the network stack, which connection a transfer uses impacts performance. Since TCP provides reliable in-order delivery, network losses in an earlier transfer can delay when the later transfers in that connection are delivered to the app.

CloudProphet currently supports two policies– "no reuse", in which each transfer uses a new connection and "no overlapping", in which a transfer can use any of the idle connections towards the destination/port. We empirically found these to be the most common patterns though others can be built as well. As with event ordering, we note that the choice of connection to use for a transfer, need not be the same as observed in the trace. CloudProphet leverages patterns, such as worker-dispatcher, to identify dependencies.

3.2 Inter-component Dependencies

So far, we have looked at how to mimic individual resource use effectively. That is, the shadows replay events in the trace and make the corresponding resource demands. Now, we focus on issues that impact the order in which events occur in the new environment.

3.2.1 Locks

Tracing locks is important, because they encode dependence between work. Literally replaying the order in which the threads gain (or block at) a lock would not work for reasons similar to those described in §2.3— in the new environment work may take different amounts of time, causing different thread orderings through locks.

CloudProphet explicitly traces the synchronous lock primitives provided by common libraries such as pthread locks, file locks, and Java monitors. For each lock operation, CloudProphet records its type (lock or unlock), the lock type (shared or exclusive), and a unique identifier for the lock (*e.g.*, its memory address). During replay, the shadows issue the same lock operations against pre-initialized lock objects. For simplicity, we ignore the performance difference across different lock implementations, and use pthread locks to replay all of them.

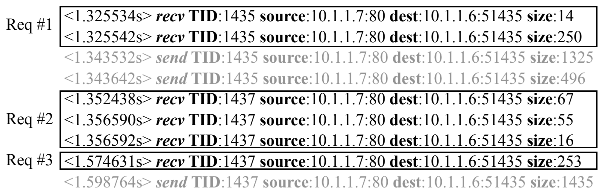
Some advanced applications (*e.g.*, databases) use low-level primitives to implement more complex locking schemes, such as locks with yielding [20] and locks with intention. Such locks are rare and replaying them requires annotating the application and hence, CloudProphet ignores them for simplicity.

3.2.2 Applying dispatcher-worker pattern

Recall in §2.2 we explained it is necessary to know the dependencies between the work (events) in order to replay them well. CloudProphet focuses on extracting dependencies from applications conforming to a common worker-dispatcher pattern. A dispatcher issues *requests* to workers who in the course of processing the request can make calls to other components across the network, invoke storage, or interact with the database.

Under the pattern, the dependencies can be captured easily based on the id of the worker thread and the time-period between receiving the request and completing the response. This leaves us with two main work items. How to identify request-response boundaries? And, how to emulate the dispatching of work? We tackle these in order.

Separating the work due to each request: By definition of the worker-dispatcher pattern, we know that requests start when a message is received by the server and responses consist of messages sent back to the clients. So, the goal here is, given a time-sorted sequence of network events, both sends and recvs along with the IDs of the threads that did them, we want to identify individual



```
Req #1 <1.325534s> recv TID:1435 source:10.1.1.7:80 dest:10.1.1.6:51435 size:14
       <1.325542s> recv TID:1435 source:10.1.1.7:80 dest:10.1.1.6:51435 size:250
       <1.343532s> send TID:1435 source:10.1.1.7:80 dest:10.1.1.6:51435 size:1325
       <1.343642s> send TID:1435 source:10.1.1.7:80 dest:10.1.1.6:51435 size:496
       <1.352438s> recv TID:1437 source:10.1.1.7:80 dest:10.1.1.6:51435 size:67
Req #2 <1.356590s> recv TID:1437 source:10.1.1.7:80 dest:10.1.1.6:51435 size:55
       <1.356592s> recv TID:1437 source:10.1.1.7:80 dest:10.1.1.6:51435 size:16
Req #3 <1.574631s> recv TID:1437 source:10.1.1.7:80 dest:10.1.1.6:51435 size:253
       <1.598764s> send TID:1437 source:10.1.1.7:80 dest:10.1.1.6:51435 size:1435
```

Figure 4: **Identifying requests from network I/O events. The sequence of *recv* events from the connection are broken into three requests based on our rules.**

request-response pairs. Note that, here, we only look at the network events triggered by clients, but not those between the server and the backend. We do so, using these rules, which are also illustrated in Fig. 4.

For each thread, an uninterrupted sequence of recvs followed by an uninterrupted sequence of sends is identified as a request-response pair. This is the common case. The thread receives a request, does some work, and then responds.

In some cases, there may be no response. So we also end a request-response pair when the same thread receives a message after a long lag (time T) from the previous receive. There is a trade-off in choosing the right T: if T is too small (*e.g.*, less than one round-trip-time), the `recvs` that belong to the same request would be erroneously inferred as a number of individual fire-and-forget requests; if T is too large, some lightweight fire-and-forget requests would be inferred as part of their following requests. In our experiments, we found that, unless the network is badly congested, the time gaps between the `recvs` of the same request are usually smaller than one round-trip-time (RTT), consistent with previous observations [44]. On the other hand, the processing time of a request is usually much larger (more than 10 folds) than the RTT within a data center. Therefore, we choose T to be $4X$ of the expected RTT.

We assign a unique ID to each identified request. During replay, the test clients (see Figure 2) send the ID of the original request that they intend to replay. Requests from the test clients are padded to be the same size as the original requests. The dispatcher issues a worker to process the request – a shadow that replays the events in the trace corresponding to the identified request-response pair.

This inference is not perfect and leads to error in some cases. For instance, if the request-response is interrupted, that is the worker receives more data from the client in the middle of sending back the response, it will be identified as two request-response pairs that are unrelated to each other. This can happen when a request is long and needs to be processed in multiple stages (*e.g.*, an acknowledgement is immediately returned by the server after receiving the request header). In some cases (*e.g.*, due to TCP retransmission timeout [40]), requests that arrive

```
while true do
    $fd = poll(fd\_set)$
    if $is\_connection\_new(fd)$ then
        $add\_fd(fd\_set, fd)$
    else if $is\_connection\_closed(fd)$ then
        $remove\_fd(fd\_set, fd)$
    else
        $req\_ID = recv\_id(fd)$
        $enqueue(req\_ID)$
    end if
end while
```

Figure 5: The pseudo-code of CloudProphet's dispatcher.

with significant inter-arrival gaps ($> T$) will also be considered to be separate and independent. These inference errors are rare for normal RPC-like applications under good network conditions. They are also unlikely to impact performance estimation much, because we still replay all the work between each inferred request-response pair. Hence we ignore them to keep the solution simple.

Emulating Dispatchers: CloudProphet implements its own dispatcher to emulate the behavior of the real application dispatcher. Figure 5 shows the dispatcher's pseudo-code. In particular, the dispatcher performs the following tasks. First, it accepts any incoming connection to the server port and adds them to a connection set (fd_set). Second, it polls for any incoming request from the connections in the set. Finally, it extracts the request's ID from the preamble and adds the ID to a request queue. The worker threads repeatedly poll new request IDs from the queue, and replay their corresponding handler events.

Several tunable parameters are available to tweak the dispatcher's behavior to better match that of the real dispatcher, such as the maximum number of worker threads and the maximum queue size. The customer can change the parameters to predict the application's behavior under different configurations.

4 Implementation

We have implemented a Linux prototype of CloudProphet, which includes three main components: tracing engine, trace analyzer, and replayer. The total lines of code are around 7.5K. In the following we describe the implementation details of each component.

Tracing engine The tracing engine is implemented as a lightweight library call interceptor through `LD_PRELOAD`. It transparently records the resource usage events from all application threads and write them to disk. Each recorded event contains four fields: a thread ID, a start timestamp, an event type, and a data field whose semantics depend on the event type.

We first intercept the libc wrappers for the I/O system calls to trace the I/O events. When intercepting these calls, the tracing engine also obtains the call context through helper functions. For instance, for a network I/O call, it would call `getsockname` over the intercepted file descriptor to obtain the local address and port. To collect database events, the engine intercepts library calls inside MySQL's native driver `libmysqlclient`. Currently, CloudProphet only supports MySQL, but it is easy to extend CloudProphet to other databases. Finally, we intercept the pthread lock, the file lock, and the Java monitor functions to trace the lock events. Note that the Java monitor functions are not strictly library calls: they are Java bytecode instructions. Therefore, we use a bytecode binary instrumentation technique [9] to intercept these Java monitor functions.

To capture computation events, the tracing engine also measures the running time of a thread between two non-computation events. This is done through the system call `clock_gettime`, which returns the virtual time of the current thread with high precision.

To reduce tracing overhead, we also build an efficient and thread-safe library to log the events to disk. The library uses per-thread event buffers to avoid global locks, and has a background thread to flush the filled buffers to disk asynchronously. We also keep all event buffers in shared memory to minimize copies.

Trace analyzer The trace analyzer is an offline program that takes the per-thread event traces as input and outputs the events for each request-response pair, following the algorithm in § 3.2.2. The trace is stored as one file per component to simplify event distribution, as all replayers of the same component can be configured to use the same trace file. In addition, an index of the handlers based on their request IDs is added at the beginning of the file for efficient lookup.

Replayer The replayer is implemented as a lightweight multithreaded server, following the design in §3. It takes a processed trace file and a configuration file as input. The configuration file includes the connectivity information towards the other replayers and the replay options. The replayer also prefetches the future events to reduce event loading overhead.

5 Evaluation

We evaluate CloudProphet in three steps. First, we show the usefulness of CloudProphet with two realistic case studies. Next, to understand why it works, we evaluate CloudProphet's accuracy over a variety of applications, and compare it with alternative designs. Finally, we evaluate the overheads of using CloudProphet.

5.1 Experiment Settings

Application Table 2 summarizes the applications we choose to evaluate CloudProphet. The applications have

App	Description	Resources used
Distributed		
RUBiS [25]	J2EE auctioning app	cpu, net, db
MediaWiki [16]	PHP collaborative app	cpu, net, disk, db
HTTP file server	File serving website	net, disk
Subversion [7]	Source version control	cpu, net, disk
TPC-C [27]	OLTP app	db
Non-distributed		
FFmpeg [13]	Video transcoding	cpu
gcrypt [26]	Cryptography suite	cpu
IOzone [14]	Disk I/O benchmark	disk

Table 2: **The applications we use to evaluate CloudProphet and the main resource types they consume.**

Cloud	Instance	CPU Model	Cores	Mem (GB)	Price $/hour
AWS	small	Intel E5507 Intel E5430	1	1.7	0.08
	medium		1	3.75	0.16
	large		2	7.5	0.32
Joyent	small	Intel E5620	1	2	0.17
	medium		1	4	0.24
	large		2	8	0.36

Table 3: **The instance types used in our evaluation. We discovered two common CPU types used in the AWS instances, but only one CPU type for Joyent instances. The prices are quoted in March 2012.**

diverse characteristics. They range from realistic web applications such as RUBiS and MediaWiki which have multiple components and consume different types of resource, to applications such as IOZone that stress one particular resource. About half of the applications are distributed, which means they involve at least a client and a server and conform to the dispatcher-worker pattern. RUBiS and MediaWiki further have three tiers on the server side: a web tier, an application tier, and a database tier. The web tier serves the incoming HTTP requests; the application tier executes business logic (RUBiS) or performs full-text search (MediaWiki); the database tier stores the application data. We further generate representative client requests to drive the distributed applications. For some applications that are originally designed as benchmarks, such as RUBiS, MediaWiki, and TPC-C, we generate requests using their official benchmark workload generators.

Cloud environments We evaluate CloudProphet inside two popular IaaS cloud platforms, Amazon AWS [6] and Joyent [15]. We choose AWS because it is the largest and most well-known cloud platform, and choose Joyent because it is relatively new with comparable service tiers as AWS. Due to budget and time constraint, we cannot cover all instance types offered by the providers. Hence, we select three instance types from each provider in the low and medium price range, as shown in Table 3. We name the three instance types from each platform according to AWS' convention as `small`, `medium`, and `large`. These instances have different configurations (CPU type, memory size, etc.) and prices. Further, according to existing measurement results [49], they also have different interference levels. We allocate all instances inside the default data center location of each provider.

On-premise environment We collect all application traces in a homogeneous on-premise cluster with 11 machines. Each machine has a quad-core Intel Xeon X3210 CPU with 4GB memory. All machines are connected with a 1Gbps LAN.

5.2 Metrics

We use CloudProphet to estimate both the performance and cost of an application. For performance, we consider two commonly used metrics: end-to-end response time and maximum application throughput. End-to-end response time is defined as the time from when a request is sent by the end-user to when the response is received. For web applications, we also consider the maximum application throughput, which is defined as the number of requests the application can process per second without exceeding a response time threshold. We set the threshold to be two seconds according to a recent study on e-commerce applications [2].

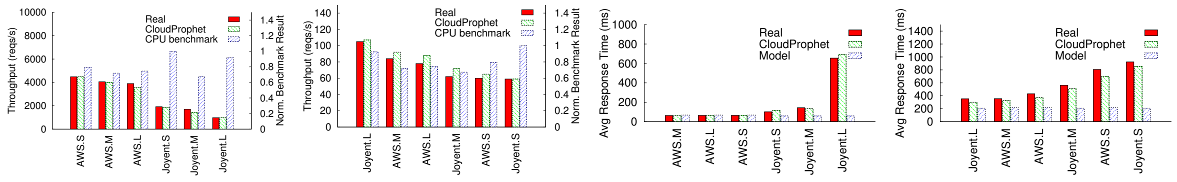
To estimate an application's performance in a target cloud environment, we first collect the traces while the application is serving representative workload on-premise. We then replay the traces inside the target cloud environment with real instances. We further adjust the replay speed to obtain the response time at different incoming rates and measure when the maximum throughput is reached.

An application's cost is computed as the total monetary cost incurred by running the application inside a cloud environment while serving expected workload. In the context of our applications, the cost includes three parts: instance cost, bandwidth cost, and database service cost. The instance cost depends on how many instances are needed and the price for each instance. The bandwidth cost depends on the amount of traffic sent across the cloud boundary and the bandwidth price. The database service cost depends on the price of the chosen database service.

We estimate an application's cost as follows. We first use CloudProphet to estimate the maximum application throughput achieved per instance in each application tier. We then compute how many instances are needed in each tier to handle the application workload. We also use CloudProphet to estimate the throughput of each database service to decide which database service to use. Finally, we measure the traffic sent by the replayers to compute the bandwidth cost.

5.3 How Can CloudProphet Help?

In this section, we use two case studies to show how CloudProphet can help customers migrate applications into the cloud.



(a) RUBiS throughput (b) MediaWiki throughput (c) RUBiS response time (d) MediaWiki response time

Figure 6: **The real and estimated maximum throughput and response time of two applications inside each candidate cloud deployment. For comparison, we also show the normalized CPU benchmarking results for throughput and the modeling results for response time.**

5.3.1 Choosing Best-Performing Cloud Environment

As mentioned earlier, a customer who plans to migrate applications into the cloud often faces the dilemma of choosing from a wide range of candidate cloud environments. One typical scenario is the customer wants to pick the best-performing cloud environments under certain budget constraint. Suppose the customer is considering the six candidate instance types in Table 3 and has a budget limit of \$3.2/hour, which could result in a small- or medium-sized deployment with 7$\sim$40 instances depending on the chosen instance type.

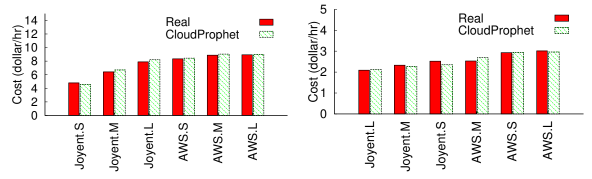
Figure 6 shows the actual throughput and response time (y-axis on the left) of running RUBiS and MediaWiki under different instance types and those predicted by CloudProphet. For comparison purpose, we also try a simple but widely-adopted method of ranking the instance types based on benchmarking results. We choose CPU benchmarks because both applications are likely to be bottlenecked by CPU. (In fact, we found other benchmarks, *e.g.*, network and storage, perform much worse.) We run the multi-threaded CPU benchmark [38] on each instance with enough threads to saturate the CPUs and measure the total number of benchmark tasks finished within a time period. We further normalize the benchmarking throughput to between 0 and 1 (y-axis on the right) because it is not directly comparable to the application throughput.

The figure shows CloudProphet accurately predicts the application throughput under all instance types (with relative error $< 16\%$). As a result, it can easily help the customer pick the best instance type (AWS.S for RUBiS and Joyent.L for MediaWiki). However, if the customer follows the benchmarking numbers, she would have picked Joyent.S for both applications which offers only roughly half of the application throughput under the best instance type.

The main reason behind the ranking differences between CloudProphet and benchmarking is because of the unpredicability of performance bottlenecks under different cloud environments. For example, in most Joyent instances the CPU utilization increases significantly under heavy network activity, possibly due to an inefficient network driver. This interference creates a convoluted bottleneck that involves both CPU and network, which cannot be captured by CPU benchmarking alone. It is also difficult to come up with the right mix of CPU and network workloads to stress such bottleneck. As another example, both the web and business tiers have high CPU utilization when running RUBiS in Joyent.L instances. When multiple bottlenecks exist, it is hard to combine the benchmarking results of individual tiers into an aggregate number that reflects the end-to-end application performance. Finally, when running MediaWiki in Joyent.S, the real bottleneck is in database instead of CPU. Without being able to predict such bottleneck shift, the CPU benchmarking result unavoidably deviates from the actual application performance. In contrast, the trace-and-replay approach of CloudProphet can naturally capture the real bottlenecks of an application running under different cloud environments.

Figure 6 also shows the response times for both applications when the incoming workload is half of the maximum throughput (around 1400 reqs/s for RUBiS and 50 reqs/s for MediaWiki). In comparison, we also estimate the response times using a modeling approach [43]. The approach models each resource at every application tier as an M/M/1 queue, and sums both the regular service time (including network latency) and the queue waiting time to estimate the total response time. It further estimates the per tier service time in the new environment through a simple CPU model that is similar to the one we used to scale the computation time (§3.1).

Again, CloudProphet attains high prediction accuracy. In contrast, the modeling approach greatly underestimates the response times, and is only accurate for the lightly loaded AWS candidates. The main reason is that the model over-simplifies both the applications and the environments. For instance, in the Joyent cloud, the model fails to capture the convoluted bottleneck described above, and hence under-estimates the utilization of the CPU resource. For MediaWiki, the model estimates the resource utilizations with reasonable accuracy, but its queueing assumption does not fit the application. This is because each application tier includes multiple servers and queues instead of just one, and



(a) RUBiS (b) MediaWiki

Figure 7: **The real and estimated cost for RUBiS and MediaWiki.**

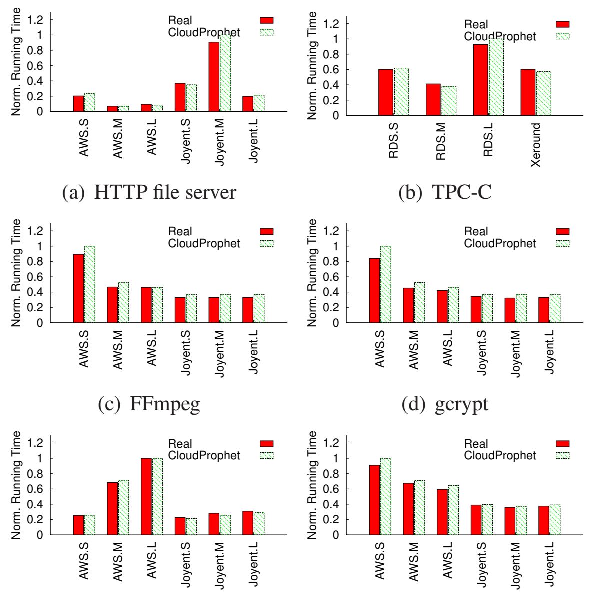
the service time at each tier is not exponentially distributed. More complex models targeted specifically at the clouds/applications would probably work better, but building such models can take significant expert effort. In contrast, CloudProphet does not abstract away the critical application and environment characteristics.

5.3.2 Estimate Cost in Cloud

Now we look at another important use case of CloudProphet, that is to help customers estimate how much their applications would cost if hosted in cloud. We first apply the CloudProphet-based approach described in § 5.2. In comparison, we also obtain the real application cost in cloud by searching for the cheapest cloud deployment that can satisfy the expected application throughput.

Figure 7 shows the estimated and real costs of the two web apps in different candidate environments. We choose the expected workload to be the same as used in the response time estimation. We found that the CloudProphet-based approach can accurately estimate the cost over all cases, and the customer can easily compare the cost-effectiveness of different cloud environments. It is interesting to note that the Joyent instance types are more cost effective than the AWS ones, despite being less efficient. This is because Joyent does not charge for bandwidth for the first 20TB transfer every month, while AWS charges around 12cents/GB for outgoing traffic. The additional bandwidth charge from AWS offsets its cost saving from instances.

Without CloudProphet, such accurate cost estimation will only be possible if we migrate the application to the target cloud environment and calibrate the cost model parameters through benchmarking. With CloudProphet, because the replay mimics the real application behavior, we can calibrate the model parameters such as the maximum throughput per instance through replaying rather than through benchmarking the actual application. Further, as the replay consumes the same amount of resource as the real application, it also incurs the same resource usage cost as the real application, such as bandwidth cost.



(a) HTTP file server (b) TPC-C (c) FFmpeg (d) gcrypt (e) IOzone (f) Subversion

Figure 8: **The real and estimated response times of the applications we evaluate. The times are averaged over 10 runs and normalized to fit between 0 and 1.**

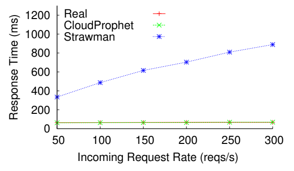
5.4 Why CloudProphet Works?

In the previous section we have shown case studies where CloudProphet can help customers pick the best-performing cloud environment and accurately estimate their applications' cost. In this section, we use controlled experiments to gain insight on why CloudProphet works. We first evaluate its accuracy with a variety of applications, and then study the impact of application dependency.

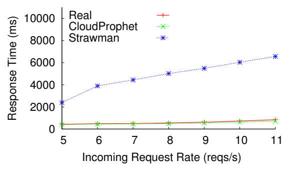
5.4.1 CloudProphet Accuracy

In previous sections, we focus on estimating the behavior of the two web applications RUBiS and MediaWiki. Now we expand our focus to all applications listed in Table 2. Figure 8 shows the real and estimated response times (or running time, if the application is not distributed) of the applications in each cloud instance type. For the database-intensive TPC-C, we test it over the different database services available inside AWS. AWS uses two different CPU models (Intel Xeon E5507 and E5430) for its instances, and we cover both by selecting E5507 for AWS.S and AWS.M and E5430 for AWS.L.

The figure shows that CloudProphet's estimation accuracy is high throughout all combinations of instance type and application. The largest relative error is 17%. This finding is surprising because the applications have very different workload characteristics: some are bottlenecked by CPUs, while the others are bottlenecked by storage or network. There are also several interesting observations. First, we found that the simple scaling approach we applied to the computation time be-

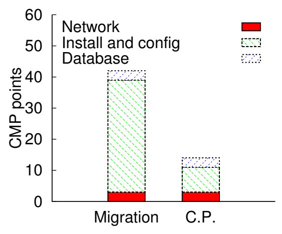


(a) RUBiS

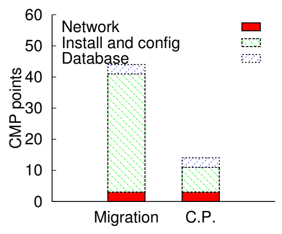


(b) MediaWiki

Figure 9: **Comparison between CloudProphet and the naive approach that replays the traces thread-by-thread. We run the experiments under different workload levels.**



(a) RUBiS



(b) MediaWiki

Figure 10: **The comparison of the migration overhead of the real applications and the deployment overhead of CloudProphet.**

Migration task	**Complexity level**		
	Low	Avg	High
Port a LAN connection	3	9	12
Port a LAN connection to WAN	3	18	27
Install and config a system software	2	6	18
Install and config a 3rd-party library	2	4	14
Populate a database table	3	4	10

Table 4: **Examples of the migration tasks modeled by CMP and their corresponding CMP points at different complexity levels.**

tween events (§3.1) works very well for practical CPU-intensive applications such as FFmpeg and gcrypt. Second, we found that CloudProphet can reproduce the storage caching effect, both for file system and database query caches. This is because the system replays the disk I/O calls and database queries under the same context as the original calls (same file location, same query schema).

5.4.2 Impact of Inter-Component Dependency

As discussed in §2.2, it is critical to extract and enforce the inter-component dependencies. We now study how important it is to follow the correct dependencies. We compare CloudProphet with a naive approach that simply replays the traces thread-by-thread, which is adopted by a number of existing trace replay work such as [36, 39, 53].

Figure 9 shows the response time estimation results for RUBiS. Here we choose one AWS large instance for each tier. We also test different workload levels, all of which are well below the maximum throughput. The response times estimated by the strawman approach are much greater than the real ones due to unnecessary blocking, as we illustrate in Figure 1. Further, the estimation error increases with the workload level, because higher workload reduces the gap between requests, which in turn causes more out-of-order arrivals. On the other hand, CloudProphet achieves high estimation accuracy. This is because CloudProphet mimics the real application dispatcher's behavior by allowing a handler to be replayed in any thread as long as its corresponding request has arrived.

5.5 Overhead

Finally, 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.

5.5.1 Computation Overhead

To evaluate the computation overhead of CloudProphet, we measure the slowdown ratio of the application when tracing is enabled. For the applications shown in Table 2, we found that the maximum slowdown ratio is 1.25 for Subversion, which triggers frequent I/O and network system calls to access its versioned database. Most other applications have their slowdown ratios close to 1, which means the tracing engine has little impact to the application performance. This is because CloudProphet does not collect very fine-grained events that require frequent instrumentation. Furthermore, the asynchronous tracing engine can effectively hide the trace-writing overhead by overlapping it with the application's other non-I/O workload.

5.5.2 Storage Overhead

We measure both the memory overhead of the tracing engine and the storage overhead to keep the traces. The memory footprint of the tracing engine is around 2MB per application thread (for the event buffer) plus a global bookkeeping overhead less than 20MB. IOzone has the highest trace storage overhead which is around 2.1MB for every tracing second, because the benchmark app always actively reads and writes the local disks, generating many I/O system calls. For the rest of the applications, the storage overhead varies from 20KB/s to 1.5MB/s. We believe such overhead is acceptable, because performance prediction typically does not require a very long trace (*e.g.*, several hours is usually good enough), in which case the trace size is on the order of several GBs to tens of GBs. Further, the incoming bandwidth is free for many clouds [15,28] to encourage migration, and uploading trace files will not incur any monetary cost.

5.5.3 Deployment Overhead

We use an existing model called Cloud Migration Point (CMP) [47] to quantify the complexity of a migration project. We choose the model because it is, to the best of our knowledge, the only one that is able to estimate the time spent on a cloud migration project with reasonable accuracy. Under the model, a migration project is divided into a number of tasks, which are further categorized into several types. Each task category has a weight that captures its time consumption. Further, for each category, three levels of complexity (low, average, and high) are defined based on task-specific characteristics (*e.g.*, the number of lines changed in a configuration task), and each level is assigned with a weight multiplier. The cloud migration point (CMP) of a migration task is computed as the weight of the task's type times the multipler of its complexity level. The total CMP of a migration project equals to the sum of all its tasks' CMPs. Table 4 shows the CMP points for a few important tasks obtained through regression [47].

We calculate the CMP points for migrating RUBiS and MediaWiki. In comparison, we also compute the points for deploying CloudProphet for the two apps. Figure 10 shows the results. We found that overall the CMP points of deploying CloudProphet are only 30% of those required to migrate the actual applications. From the broken-down results, we found that the main difference comes from the installation and configuration tasks. The other two types of task (network connection and database migration) are lightweight because the connections remain the same type and there is no change of the database schema and query type. This confirms our discussion in §2.1 that software installation and configuration can introduce significant overhead. In contrast, CloudProphet requires only one software package (the replayer) and a minimum amount of configuration (a few lines to indicate the IP addresses of the other replayers).

6 Related Work

In the cloud space, understanding the performance and cost implication of cloud migration has received growing attention. Hajjat *et al.* proposed an optimization framework to decide which application components to migrate to the cloud [34]. It requires instrumenting the application components in the cloud to obtain key model parameters. CloudProphet complements such framework as one can use CloudProphet instead of the actual application to calibrate the parameters, saving the migration overhead. Tak *et al.* qualitatively analyzed several key factors that can impact the application cost in the cloud [46]. CloudProphet can quantitatively evaluate both the performance and cost.

Benchmarking is a common approach to assess an application's performance in a new environment [11, 38]. However, finding a representative benchmark that behaves similarly to the real application is difficult, because the benchmarks are inherently built for simplicity. CloudProphet uses resource usage-perserving shadows to estimate the real application's behavior. This can be seen as building an application-specific benchmark by automatically tracing the application.

CloudProphet is also related to existing work on disk I/O trace and replay [36, 39, 53]. Similar to CloudProphet, this work also collects application events at different abstraction levels and replays them with a new storage sub-system or a different network implementation. CloudProphet differs from them in two aspects. First, CloudProphet targets at dispatcher-worker applications and uses a special shadow to mimic the dispatcher's behavior, while the existing work either assumes deterministic per-thread workload [36,39] or uses heuristics [53] to approximate the behavior. Second, CloudProphet covers a much broader range of resource types (though with an eye for practicality), while the existing work focuses on I/O events.

The dispatcher-worker application pattern has been previously leveraged to extract the request processing paths inside a multi-tiered service [45]. CloudProphet goes beyond extracting the processing paths, as it also extracts the resource usage to handle each request and replays them in the new environment.

7 Final Remarks

Easily estimating the behavior (performance, cost, scaling) of an application in a new environment can be a powerful tool. Such a tool can help app owners choose, from among the plethora of available cloud platforms, the one that is best suited for their application. It can also help app architects quickly examine the trade-offs with various design choices.

We presented CloudProphet, a system that achieves this goal by replaying the work with app-specific shadow programs. CloudProphet generates these shadows by tracing the application, ensures that the shadows exercise the same resources as the original application and leverages the popular dispatcher-worker pattern to preserve dependencies across components. Shadows are simple code that can trivially be ported across environments.

We demonstrated the usefulness of CloudProphet in helping customers migrate their applications to the cloud and show CloudProphet to be accurate for a variety of applications.

References

[1] 2010 state of the data center global data. http://www.symantec.com/content/en/us/about/media/pdfs/Symantec_DataCenter10_Report_Global.pdf.

[2] Akamai reveals 2 seconds as the new threshold of acceptability for ecommerce web page response times. http://www.akamai.com/html/about/press/releases/2009/press_091409.html.

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

[4] Amazon Simple Queue Service. http://aws.amazon.com/sqs/.
[5] Amazon SimpleDB. http://aws.amazon.com/simpledb/.
[6] Amazon Web Service. http://aws.amazon.com.
[7] Apache Subversion. http://subversion.apache.org/.
[8] Apache Thrift. http://thrift.apache.org/.
[9] BTrace. http://kenai.com/projects/btrace.
[10] Chef. http://www.opscode.com/chef/.
[11] CloudHarmony. http://cloudharmony.com/benchmarks.
[12] 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.
[13] FFMpeg. http://ffmpeg.org.
[14] IOzone Filesystem Benchmark. http://www.iozone.org.
[15] Joyent Cloud. http://joyent.com.
[16] MediaWiki. http://www.mediawiki.org.
[17] Microarchitecture-independent workload characterization studies using pin. IISWC-07 Tutorial.
[18] Microsoft SQL Azure. https://www.windowsazure.com/en-us/home/features/sql-azure/.
[19] Microsoft Windows Azure. http://www.microsoft.com/windowsazure.
[20] MongoDB: how does concurrency work. http://www.mongodb.org/display/DOCS/How+does+concurrency+work.
[21] Protocol Buffers. http://code.google.com/p/protobuf/.
[22] Puppet. http://puppetlabs.com/.
[23] Rackspace Cloud. http://www.rackspacecloud.com.
[24] Rackspace Cloud Servers versus Amazon EC2: Performance Analysis. http://www.thebitsource.com/featured-posts/rackspace-cloud-servers-versus-amazon-ec2-performance-analysis/.
[25] RUBiS: Rice University Bidding System. http://rubis.ow2.org/index.html.
[26] The Libgcrypt Library. http://www.gnupg.org/documentation/manuals/gcrypt/.
[27] TPC-C: On-line Transaction Processing Benchmark. http://www.tpc.org/tpcc.
[28] Web Hosting in Amazon AWS. http://aws.amazon.com/web-hosting.
[29] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. *UCBerkeley, TR. UCB/EECS-2009-28*, 2009.
[30] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebooks photo storage. In *OSDI'10*, 2010.
[31] 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.
[32] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. In *NSDI*, 2007.
[33] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, Jan. 2009.
[34] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In *SIGCOMM*, 2010.
[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*, 2005.
[37] E. Koskinen and J. Jannotti. Borderpatrol: isolating events for black-box tracing. In *Eurosys*, 2008.
[38] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Internet Measurement Conference*, 2010.
[39] 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.
[40] V. Paxson, M. Allman, H. J. Chu, and M. Sargent. Computing tcp's retransmission timer. RFC6298, 2011.
[41] B. H. Sigelman, L. Andrarroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.
[42] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Eurosys'07*, 2007.
[43] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A dollar from 15 cents: cross-platform management for internet services. In *USENIX ATC*, 2008.
[44] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, 2005.
[45] 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.
[46] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam. To move or not to move: The economics of cloud computing. In *HotCloud*, 2011.
[47] V. T. K. Tran, K. Lee, A. Fekete, A. Liu, and J. Keung. Size Estimation of Cloud Migration Projects with Cloud Migration Point (CMP). In *ESEM*, 2011.
[48] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS*, 2005.
[49] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *IEEE INFOCOM*, 2010.
[50] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, and S. K. C. Kim. Profiling network performance for multi-tier data center applications. In *USENIX NSDI*, 2011.
[51] G. Zhang and L. Liu. Why do migrations fail and what can we do about it? In *USENIX LISA*, 2011.
[52] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. Justrunit: Experiment-based management of virtualized data centers. In *USENIX ATC*, 2009.
[53] N. Zhu, J. Chen, and T.-C. Chiueh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *FAST*, 2005.