PseudoApp: Performance Prediction for Application Migration to Cloud

Byung Chul Tak Chunqiang Tang Hai Huang Long Wang
IBM T. J. Watson Research Center
Hawthorne, NY 10532, USA

***Abstract*—To migrate an existing application to cloud, a user needs to estimate and compare the performance and resource consumption of the application running in different clouds, in order to select the best service provider and the right virtual machine size. However, it is prohibitively expensive to install a complex application in multiple new environments solely for the purpose of performance benchmarking. Performance modeling is more practical but the accuracy is limited by system factors that are hard to model. We propose a new technique called *PseudoApp* to address these challenges. Our solution creates a pseudo-application to mimic the resource consumption of a real application. A pseudo-application runs the same set of distributed components and executes the same sequence of system calls as those of the real application. By benchmarking a simple and easy-to-install PseudoApp in different cloud environments, a user can accurately obtain the performance and resource consumption of the real application. We apply PseudoApp to Apache and TPC-W and find that PseudoApp accurately predicts their performance with 2-8% error in throughput.**

I. Introduction

Cloud is no longer an option that organizations can afford to overlook regardless of their sizes or business area. Broadly speaking, migration into a cloud can be done in two ways. One approach is to develop new applications tailored specifically to the available cloud features, e.g., auto scaling and high availability in Platform-as-a-Service (PaaS) [5], [6], [9], [18]. This approach, however, has high up-front costs and long time-to-value cycles. Another approach is to migrate existing legacy applications into the cloud, which is the focus of this paper.

Legacy application migration is often time consuming and error prone. There is a significant risk of encountering unknown issues and missing the project deadline, especially for distributed applications that involve multiple heterogeneous components. It was reported that even migrating the relatively simple and supposedly well-documented Java PetStore benchmark to a cloud took more than 22 and 36 hours for preparation and migration, respectively [17]. From our own experience in a previous project, it took about a month to set up a solution that involved half a dozen different products. Another even bigger challenge is that a legacy application may have gone through many undocumented changes over a long period of time. It is impossible to quickly reproduce these changes on a fresh installation.

We further observe that the migration effort is often dominated by the pre-migration assessment rather than the actual migration. Tran et al. [17] reports that selecting proper cloud providers and server types requires significant effort during preparation for migration. Suppose an organization wants to assess which cloud out of three candidate clouds is the best fit (e.g., in terms of performance and/or cost) for each of its n applications, it has to perform a total of $3n$ migrations in order to complete the assessment. Suppose 50% of the applications are eventually considered not a fit for any cloud (i.e., only $n/2$ of the applications will be migrated), the number of pre-migration assessments is $3n$ — a 500% overhead!

Our approach, *PseudoApp*, addresses this problem by providing a quick way of assessing an application's performance (e.g., throughput and response time) and cost (e.g., due to VM size and disk/network traffic) in a potential target cloud, without actually migrating the real application. Our approach consists of the following steps:

1) Profile the application in its legacy environment to extract its resource consumption behavior at the system call level.
2) Automatically construct what we call the PseudoApp components that mimic the real application's resource consumption behavior.
3) Install the PseudoApp components in a cloud. This is a trivial task because the PseudoApp component is simple and has no environment dependency. For example, even if the real application requires a particular library version or has an obscure parameter hidden in a configuration file, the PseudoApp component has no dependency on those.
4) Generate workloads for the PseudoApp components, and benchmark their performance and resource consumption, as if they were the real application.

Figure 1 shows the thread and system call level structure of a distributed application discovered through profiling. The PseudoApp component has the exact number of components and topology as the real application does. When processing a request, the PseudoApp component goes through exactly the same request-processing path and produces the same system call invocation sequence as that in Figure 1. The only difference is that, the real application does real work, whereas the PseudoApp component performs CPU spinning, writes meaningless data to files, and sends network messages with meaningless payloads.

The PseudoApp approach offers several advantages. First, it allows for a quick performance and cost assessment in a new environment, without going through the painful pro-

978-3-901882-50-0

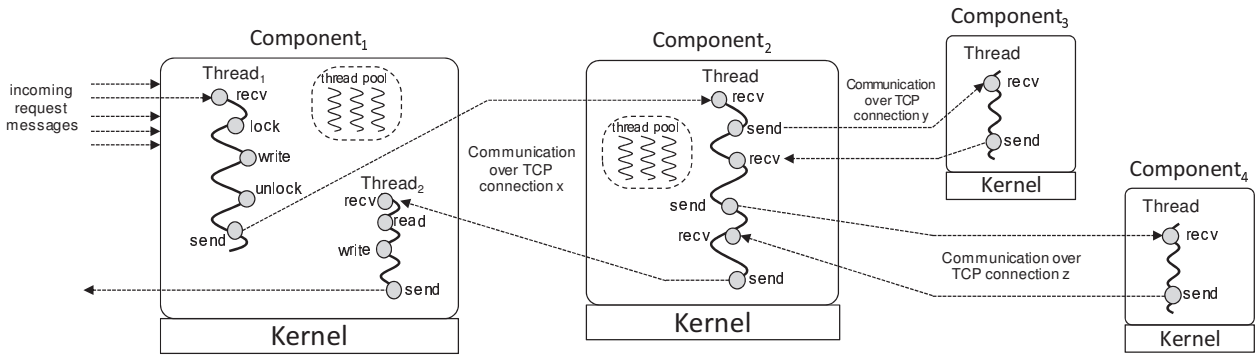


Fig. 1. Anatomy of a distributed application from the PseudoApp perspective. Component$_1$ receives multiple external requests at the same time and this figure shows only the end-to-end execution flow of one request. Other requests are processed in parallel. Component$_1$ depicts an example case where the request processing involve different threads at different time. Component$_2$ shows one example case where a single thread handles one request. System calls are invoked along the request-processing path, e.g., file read/write and network send/recv.

cess of application installation and configuration. Second, its performance result automatically reflects the idiosyncrasies of the new environment, which are often hard to capture in a pure model-based approach. For example, factors such as contention in shared network/storage and the scheduling policy automatically plays in when running the PseudoApp components. Finally, it provides performance insights for scenarios that are hard to evaluate even if we are able migrate the real application. For instance, the effect of changing the application topology can be evaluated by easily restructuring the PseudoApp component's topology, e.g., adding one more virtual machine to the middle tier.

The rest of this paper is organized as follows. In Section II we describe the key ideas of PseudoApp as well as details of current prototype implementation. Next, in Section III we present the evaluation results of PseudoApp when applied to various applications and environments. Related works are described in Section IV. Finally we make concluding remarks in Section V.

II. Our Approach: PseudoApp

We hypothesize that resource consumption and synchronization delays are the deciding factors of application performance. In Figure 1, *PseudoApp* components (which comprises one 'pseudo-application' together) precisely reproduces these factors along the end-to-end processing path of each individual request for different types of requests exhibited by the application.

When benchmarking (i.e. measuring the performance of) the pseudo-application, the workload generator sends requests to Component$_1$ with the desired concurrency level. Each request message begins with a request type ID, which instructs the components to replay a particular processing path. A precise replay of the system call sequence is also important. For example, the following two sequences may seem to consume the same amount of total resources, but their performance results can be very different:

1) Consume CPU for 100 ms and then perform 10 disk reads back to back.
2) Consume CPU for 10 ms and then perform one disk read. Repeat this pattern 10 times.

A. Overview

We adopt the vPath [16] technique to extract the application structure and path information as depicted in Figure 1. vPath reconstructs request-processing paths by observing system call events in a black-box manner and reasoning about their causality. A key observation is that an incoming message is typically assigned to a thread (e.g., Thread$_1$ of Component$_1$ in Figure 1) from a thread pool. The thread processes the request until it triggers another outgoing message.[1] This implies that, within one component, we can use the IDs of threads that invoke system calls to link together system calls on the same request-processing path.

vPath extends the request-processing path across components by pairing up a `send` system call of one component with a `recv` system call of another component on the same TCP/UDP connection. This task is equivalent to finding all the dotted arrows between components in the Figure 1. A TCP/UDP connection is uniquely identified by the endpoints $(src\_ip, src\_port, dest\_ip, dest\_port)$. In summary, vPath reconstructs end-to-end request-processing paths by combining two causality reasoning techniques that leverage thread IDs and network endpoints.

vPath only monitors network-related system calls. We extend it to 1) monitor file-related system calls (for replaying disk I/Os), 2) record context-switch events and timestamp of system calls (for replaying CPU busy time), and 3) track synchronization events such as `pthread_mutex_lock` (for replaying synchronization delay). The recorded events form the application's *profile*.

A pseudo-application is a distributed execution engine that replays events according to the profile of the real application. Consider the example in Figure 2. We deploy two pseudo components, C_1 and C_2. A workload generator (not shown

[1]This is a simplified example. We extend vPath to handle more sophisticated program patterns, including event-driven programs. Details are omitted here.

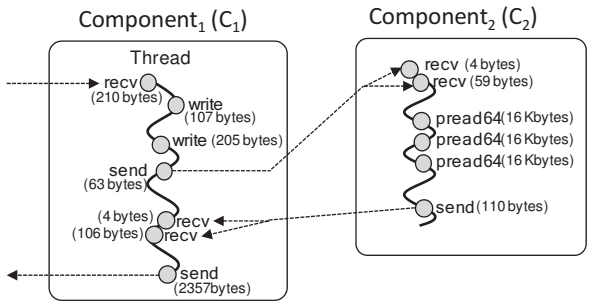


Fig. 2. A simplified request-processing path example taken from a real application. The CPU busy time between system calls are traced but not shown in the figure for the sake of brevity.

in the figure) sends a 210-byte message to C_1. The message contains a request type ID. C_1 finds in the profile the trace that matches with the request type ID, and invokes the system call sequence: *write*, *write* and *send*. Between these events, it inserts CPU busy-loops to match the CPU consumptions recorded in the profile. C_2 receives a 63-byte message from C_1 in two `recv`, replays events recorded in the profile, and finally returns a message back to C_1. C_1 finishes the rest of the processing and returns a 2,357-byte message, filled with random data.

It should be emphasized that the pseudo-application only replays the amount of resource consumption and the sequence of events, but does not enforce any specific timing or scheduling constraints. The time it takes to consume a specific amount of resource or how two concurrent threads are scheduled are dependent on the characteristics of the target environment. PseudoApp induces this behavior naturally because that is how the real application would run if it were deployed to the new environment.

B. Handling of Different Resources

This section presents a detailed discussion on how to handle different types of resources.

1) CPU Profiling and Replay: PseudoApp uses a busy-loop with integer computation to emulate the real application's consumption of δ CPU cycles. In order to determine the correct number of loops to spin in the target environment, we perform a CPU profiling with a small program that loops n times. It also records the number d of elapse CPU cycles. To consume δ CPU cycles, PseudoApp repeats the integer loop $\delta/(d/n)$ times. This method may introduce inaccuracy. The CPI (Cycle per Instruction) of the real application can vary depending on L1/L2/L3 cache, memory, and other microarchitecture factors. It is a subject of future work to more faithfully replay a certain instruction mix.

2) Network and Disk I/O: PseudoApp replays the real application's I/O activities by invoking the same sequence of system calls with the same I/O sizes for each request, but writing random data to disk or network. For a disk I/O, the data location significantly impacts the performance. During profiling, PseudoApp tracks the file offset of disk I/Os. During replay, PseudoApp can either repeat the exact same file offset history, or generate file offsets from a probability distribution that is modeled after the real trace. There are pros and cons of each approach. Our current implementation uses the second approach. A comparison of the two approaches is a subject of future work.

3) Memory Access: The number of active memory pages (i.e. Resident Set Size) affects the performance in the following way. OS kernel usually utilizes all available memory pages that are not assigned to processes as page cache. To induce a faithful page cache size, which significantly affects disk I/O performance, a PseudoApp component periodically touches the same number of active memory pages as that observed during profiling, which helps prevent them from being swapped out. Besides the size of active memory, memory access pattern also affects performance, e.g., due to hit/miss in L1/L2/L3 cache. Currently PseudoApp does not trace or replay memory access patterns because of the high overhead in doing that. It is left as a future work to study how to efficiently summarize and reply memory access patterns, e.g., leveraging reuse distance analysis [3].

4) Thread Synchronization: Synchronization can have significant impact on application performance, reducing throughput and increasing response time. Currently, PseudoApp profiling is implemented in the kernel. It monitors the invocations of `futex` system calls during profiling and recreates the same effect by inserting `pthread_mutex_lock` and `pthread_mutex_unlock` in the pseudo-application. One potential problem is that not every application-level synchronization operation can be observed in the kernel during profiling, because on Linux, `pthread_mutex_lock` is implemented in such a way that only a real contention results in a `futex` system call. If there is no contention, it is simply a library call, which is invisible to the kernel. This is an implementation issue of PseudoApp. We are actively converting PseudoApp to perform profiling at the user level, by using LD_PRELOAD to intercept library calls, which would resolve this problem. In practice, we do not observe the current limitation dramatically impacts the performance. See Section III-B for more details.

C. Some Implementation Details

Most of our engineering efforts in PseudoApp are put into the application profiling, which can be done in several different ways, depending on the environment constraints such as OS types, virtualized or physical environment, and the software level (e.g., libc, guest kernel, or hypervisor) to instrument profiling. Our current implementation assumes a Xen-based environment, where applications run in VMs. We modify the guest kernel to monitor system calls and context switches. At the entry and exit point of a system call we monitor, the system call parameters and return values are captured and transferred to the Xen hypervisor through hypercalls. The hypercall handler in the Xen hypervisor invokes `xentrace` to store the information in log files. Although our prototype uses `xentrace`, it is for convenience rather than mandatory, and can be replaced by any available kernel logging mechanism. The

system calls traced by PseudoApp include `read`, `write`, `recv`, `send`, `bind`, `connect`, `listen`, `accept`, `time`, `futex`, etc.

Our profile analyzer reads the log files created by `xentrace`, and follows the principles described in Section II-A to extract the request-processing paths and save them in a profile. We then deploy the pseudo-application in a cloud together with the profile. The profile controls the behavior of the pseudo-application. During benchmarking, our PseudoApp workload generator submits requests at a controlled concurrency level to the front-end component of the pseudo-application. The workload generator and the front-end component communicate through a pre-defined message format. Upon receiving a message, a pseudo-application component executes the corresponding system call sequences and CPU busy time between system calls, both of which are available from the stored profiles. During the execution, one component of the pseudo-application may send a message to another component. The receiving component replays the corresponding system call sequence and CPU busy time from the profile. As the request-processing travels through the distributed PseudoApp components, every component performs replay faithfully according to the profile.

III. Evaluation

We use PseudoApp to predict the performance of real applications in multiple clouds: our Xen-based in-house private cloud and the Amazon EC2 public cloud [1]. We compare the response time and throughput predicted by PseudoApp with those measured from the real applications running in the clouds, in order to evaluate the prediction accuracy. The applications include the Apache web server and the TPC-W benchmark developed at New York University [12]. TPC-W is a fully J2EE compliant application. We use a two-tier configuration for TPC-W: a front-end JBoss tier (JBoss 3.2.8SP1) and a MySQL database tier (MySQL 4.1).

Experiments with Apache allow us to control various resource conditions so that we can see how well PseudoApp is in mimicking the application performance with consumption of individual resource types such as CPU, network I/O, and disk I/O, separately. Experiments with the TPC-W demonstrate the PseudoApp performance in predicting a typical e-commerce application.

In summary, the experiments validate that the PseudoApp approach is effective.

- PseudoApp accurately predicts the performance of Apache in the presence of various resource conditions (saturation of network, CPU, and disk I/O) and system parameters such as file sizes.
- PseudoApp closely follows the performance of TPC-W over varying workload intensity and various system settings with an average of 3.0-3.8% prediction error for throughput and 5.3-8.5% error for response time.

Settings	Response Time Error	Throughput Error
(Case1) Network Saturation	12.9%	1.4%
(Case2) Similar to Case1, but smaller web page size	18.3%	3.0%
(Case3) CPU Saturation	19.6%	7.6%
(Case4) Disk I/O bandwidth Saturation	6.8%	6.1%
TPC-W Setting 1 (In-house server, virtualized)	8.5%	3.8%
TPC-W Setting 2 (Amazon EC2 2 VMs)	5.3%	3.0%

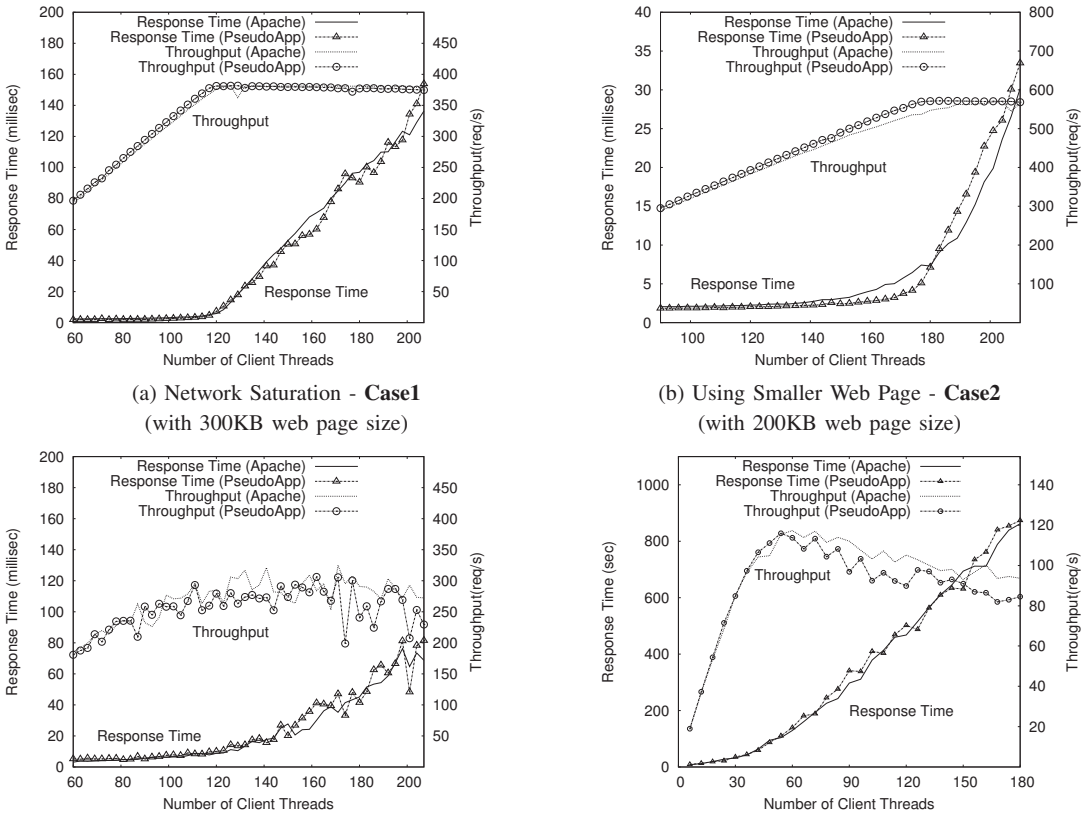
TABLE I
Average Error of Performance Prediction.

A. Evaluation Using Apache Web Server

In our experiments, each server machine used in the evaluation has dual Intel Xeon 3.4GHz (2048KB cache) processors with hyperthreading. Memory size is 1GB and the network bandwidth is 1Gbps. Profiling of the Apache web server is done within a VM that runs in a virtualized environment of Xen 3.1.4 on a physically different set of hardware (Intel Xeon 3.06GHz, 512MB cache, different networking H/Ws etc.) from where we conduct the following evaluations. Obtained profiles are, then, used to construct the PseudoApp components for Apache. These PseudoApp component and Apache are deployed to the same hardware (previously mentioned Intel Xeon 3.4GHz, 2048KB cache servers) that is also virtualized, for gathering performance comparison results.

Our experiments generated the following four different resource conditions.

- **[Case1] Network Bandwidth Saturation**: One VM is given one physical CPU. The network bandwidth between the workload generator and the Apache VM is 1 Gbps. The Apache web server serves a static web page of 300 KB size. The workload generator repeatedly accesses the same web page to consume CPU and network resources, excluding the effect of storage access in this configuration.
- **[Case2] Network Saturation with Different Web Page Size**: All the conditions are identical to the **Case1** except that the web page size is reduced to 200KB. This case is intended to show what happens to the network bottleneck if the web page size is reduced (thereby reducing the network usage which may shift the bottleneck from network to CPU).
- **[Case3] CPU Saturation**: In this setting, we launch two VMs that share a single physical core. One VM runs an infinite integer loop that uses up all the allocated CPU. Another VM runs the Apache web server. The CPU allocation ratio of the integer-loop VM to the Apache VM is 2:1. This configuration makes CPU the bottleneck resource. If the CPU is given entirely to the Apache VM,



(a) Network Saturation - **Case1** (with 300KB web page size)

(b) Using Smaller Web Page - **Case2** (with 200KB web page size)

(c) CPU Saturation - **Case3**

(d) Disk I/O constrained Setting - **Case4**

Fig. 3. Performance comparison of Apache and PseudoApp for cases **Case1, 2, 3** and **Case4**. Each graph is averages of between 5 to 10 runs.

the network would become the bottleneck. Apache serves a 300KB static web page, as in the previous setup. Two VMs' CPU allocation ratio is 2:1 (1 for the Apache VM). This CPU sharing configuration is devised in order to make the CPU the bottleneck resource. If full single CPU is given to the Apache VM, network bandwidth becomes the bottleneck in our hardware configurations. Again, the Apache web server serves 300KB-sized static web page.

- **[Case4] Disk Bandwidth Saturation**: This setting is designed to evaluate PseudoApp's capability of reproducing the application behavior under a disk-I/O intensive workload. We prepared a set of 500 web pages for Apache, closely following the reported properties of real web sites [2]. The average web page size is set to 800KB, with a lognormal distribution. The total size of the web pages (460MB) is made close to the Apache VM's total memory size (512MB) in order to force page cache miss and trigger actual disk I/Os. For each web request, the workload generator randomly selects a web page to access.

The PseudoApp component for Apache is created with profiling information obtained under low workload intensity. We did not perform the profiling under heavy workload in order to fine-tune the PseudoApp component. All the workload intensities shown in the result graphs are not seen during the profiling phase. This demonstrates PseudoApp's advantage in making accurate prediction for previously unseen, new workloads.

Figure 4 shows how Apache works internally when processing a web request. Fetching a web page from the apache web server generates the sequence of system calls as shown in Figure 4 (b). The number of `sendfile64` depends on the total web page size and the graph in Figure 4 (a). The number of bytes each `sendfile64` sends are not equal. It has a ramping-up period and never exceeds 1.5 MB per invocation. Our PseudoApp components for apache are built to follow all of these properties.

The results for the settings **Case1**, **Case2**, **Case3** and **Case4** are presented in Figure 3. The average error of PseudoApp on the Apache is summarized in Table I. In Figure 3 (a), which is the case of network saturation, Apache's response time starts to increase linearly as the number of concurrent clients goes beyond 120. At that point, the network is fully saturated. The throughput curve tops at 381 reqs/sec, which is close to the theoretical upper bound of the 1 Gbps network: 300KB$\times$381req/s$\times$8bits=0.93Gbps. Although not shown in the figure, the CPU utilization is consistently below 50%. Figure 3a) shows that PseudoApp's response time and throughput

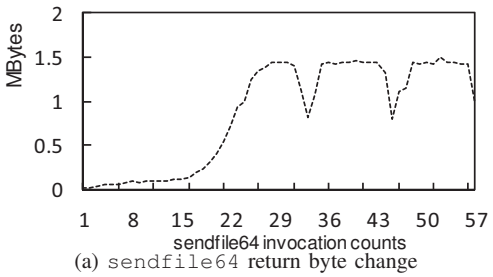
closely match those of the real Apache server.

Network saturation of **Case1** is the consequence of the web page size being 300KB. What would happen to the bottleneck if the web page size is reduced? We can expect that the network bandwidth capacity would allow higher throughput. But, would CPU become the bottleneck before the network bandwidth limit is reached as we increase the throughput? **Case2** is designed to find out the answer to this by reducing the file size to 200KB. According to Figure 3(b), it turns out that the bottleneck does not shift from network to CPU. The throughput tops at about 580 req/s reflecting the effect of smaller web page size, and the response time is also significantly smaller than **Case1**. PseudoApp's result follows all of these behaviors well. Part of the reason why the network remains as the bottleneck is because the CPU consumption is reduced together as the web page size gets smaller. The answer to the posed question is not immediately clear and also difficult to answer based on intuitions. However, PseudoApp successfully demonstrate that it delivers accurate answers to these what-if questions.

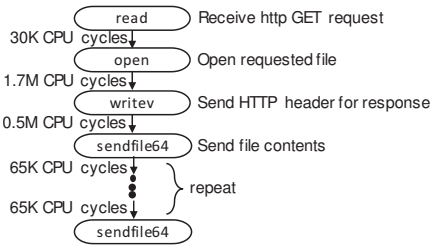
The setting **Case3** is designed to observe how effectively PseudoApp can reproduce the performance under the CPU saturation condition. CPU saturation is achieved by having two VMs share a core and, at the same time, by lowering the CPU scheduling weight of the Apache VM. Comparing with **Case1**'s results, the response time graph shows nonlinear curvy increasing trend and falls under the response time graph of **Case1**. Plateau of the throughput graph lies below that of **Case1** indicating network is not a bottleneck. We can see that the CPU saturation has less impact to the response time than the network saturation does, but it degrades the throughput more. Jaggedness of both the response time and throughput is partly due to the VM scheduler's characteristics. PseudoApp is able to reproduce all of these performance characteristics.

We also conducted experiments to study the PseudoApp's capability in reproducing the performance under disk I/O intensive workloads using the setting **Case4**. Figure 3 (d) compares the performances of Apache web server and the corresponding PseudoApp component. Note that the scale of response times is one order higher than others due to the latencies of actual disk I/Os. As a preparation step before the actual run, the PseudoApp component created the same number and size of dummy web pages as described earlier. We can see that PseudoApp technique is able to portray the actual performance in case of heavy disk I/O workloads as well.

From these evaluations using Apache web server we verify the PseudoApp technique's efficacy in reproducing the performance for individual resource types. The results support our hypothesis that mimicking the resource consumption (and without enforcing any other execution rules such as specific timing or orders) is a promising way of reproducing the performance. We have also seen that profiling information at one workload level is sufficient for creating a solid PseudoApp component.



(a) `sendfile64` return byte change



(b) System call sequence of HTTP GET

Fig. 4. Apache's behavior of static file fetching requests. The return byte sizes of repeated sendfile64 are not uniform. The size increases with the pattern shown in (b) until it reaches 1.4 MB per invocation.

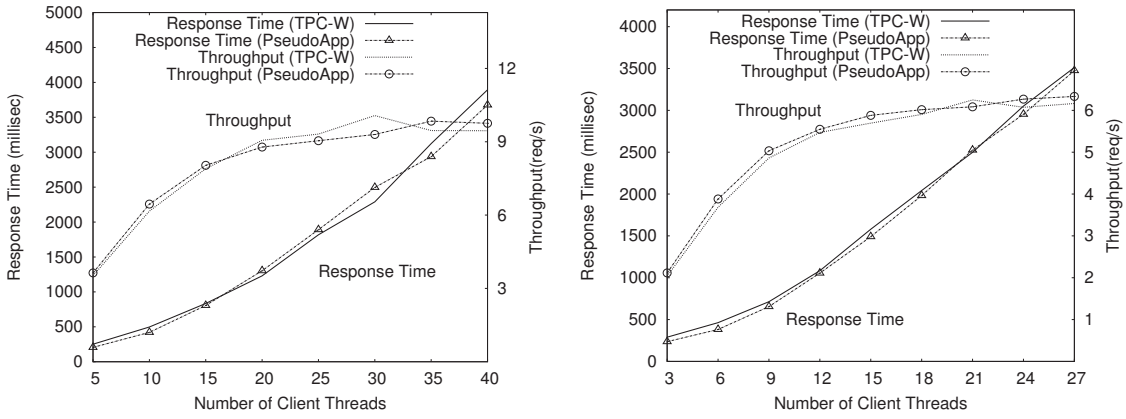
B. Evaluation Using TPC-W E-commerce Benchmark

In this experiment, we compare the measured performance of the real TPC-W with the prediction given by PseudoApp. We created two PseudoApp components: one for JBoss and the other for MySQL. Our evaluation was done in two different cloud environments:

- Setting 1: In-house private cloud, where two VMs are created on one physical server. One VM runs JBoss and the other runs MySQL. The hardware configuration is identical to that of the Apache experiments. Each VM has its own dedicated core.
- Setting 2: Amazon EC2, where two small instances are created. One VM runs JBoss and the other runs MySQL. Each VM is configured with 1.7 GB memory and 1 EC2 compute unit.

PseudoApp was able to produce similar performance results as the real TPC-W, over varying workload intensities. Figure 5 (a) compares the performance of TPC-W and PseudoApp in our private cloud. Although the CPU utilization changes are not presented here, the CPU utilization of MySQL reaches 70-80% as the number of client threads exceeds 15. Even under such an overloaded condition, the performance of PseudoApp closely follows that of the real TPC-W.

Figure 5 (b) shows the results in Amazon EC2. We purchased three small EC2 instances and used two of them for hosting TPC-W (JBoss and MySQL), and used the third one for workload generation. The performance of TPC-W and PseudoApp matches closely even under a heavy load, as evidenced by the flattened throughput and the steep rise of response time.



(a) Setting1: In-house private cloud (b) Setting2: Amazon EC2

Fig. 5. Response time and throughput comparison of TPC-W and PseudoApp in two different environment.

System Call	Frequency	Total Cycles Consumed	ms per system call
time	511256	495M	0.00028 ms
pread64	73780	11999M	0.047 ms
read	51694	558585M	3.12 ms
rt_sigprocmask	27120	82M	0.0008 ms
futex	23349	1969065M	**24.37** ms
llseek	15759	15M	0.0002 ms
fcntl64	13434	21M	0.0004 ms
sched_setscheduler	12488	42M	0.0009 ms
write	6305	507M	0.023 ms

TABLE II
TURN-AROUND TIME OF TOP 9 SYSTEM CALLS DURING THE BUSIEST TIME OF MYSQL EXECUTION IN TPC-W.

Our prediction error metric is defined as $E = (R_{tpcw} - R_{pseudoapp})/R_{tpcw}$, where R is either throughout or response time. For both our private cloud and the EC2 public cloud, the errors were 1-17% and 1.4-5.3% for response time and throughput, respectively.

One important factor that affects the performance in the overloaded region is the thread synchronization operations via `futex` system call. We confirm this by counting each system call's occurrences and measuring the latencies when running the real TPC-W. Table II lists top-9 most frequently invoked system calls during a highly overloaded 15-second period of MySQL. The forth column is the time (in millisec) spent by the system calls in the kernel mode starting from the entry point of the system calls to the exit point. Although `time` system call is invoked most frequently, it is not the major contributing system call to the overall response time since the turn-around time of the individual call is very small (.00028ms).

The largest contributor turns out to be the `futex` system call, with an average latency of 23.47 ms for each invocation. This latency is the duration of time between when a thread is put into the wait queue due to mutex conflict and the time when it is awaken and exits the `futex` call. Without emulating `futex`, PseudoApp's response time is too optimistic, only about half of the real TPC-W's response time. Therefore, accurately capturing thread contention is critical for PseudoApp.

We collected information in Table II only for the purpose of understanding the impact of different system calls. It is not needed for constructing PseudoApp, as our tool fully automates the process of PsedudoApp construction, by using the augmented vPath to profile TPC-W under a light workload.

IV. RELATED WORK

Although there exist numerous research works on performance modeling [4], [11], [13], [15], many of them are not directly applicable to the problem of performance prediction for a disparate environment. This is because many performance models are often tightly coupled to the environment properties that the application is currently running. If the environment changes, the models have to be refined and tuned again using the parameters of the new environment which may not always be possible.

Cross-platform performance prediction has been studied in the context of processor architecture designs. The motivation is to build a platform-independent performance model so that it can quickly explore the processor design space. Some of them propose to use machine learning techniques [8] or statistical regression [10]. In the work of Hoste et al. [7], performance of an application is predicted by first collecting performance scores of known benchmark, and second, profiling the new application, and finally, selecting the closest performance from the space based on the similarity measures. The inherent problem is that the accuracy is limited if new application is not similar to any of the benchmarks. Also, their technique requires instrumenting the application. In the context of processor architecture, Ipek et al. [8] have used artificial neural networks to build the performance model.

Stewart et al. used so-called *trait models* [14] in performance prediction. Each trait model describes one aspect of the system, e.g., cache miss rate for a given processor cache size, or the total number of instructions for a given workload. Multiple trait models are combined into one formula to form the final performance model. This approach requires intimate

knowledge of the application's internals in order to compose the final performance model from the trait models.

Justrunit [19] learns the performance of application components by actual experiments rather than from the models. It assumes a virtualized environment where components reside in separate VMs. In order to learn the relationship between VM's resource allocation and performance in a live production system, they propose to create a clone of target components and sandbox them. Actual traffic is duplicated and fed into the sandboxed VMs that have different resource allocations. All the input and output traffics are controlled by the in&out proxies to ensure correctness of the execution. However, there are applications whose correctness cannot be guaranteed by the in&out proxies. Hence Justrunit is not a general method.

V. Conclusion

As cloud becomes indispensable, migrating legacy applications from a traditional hosting environment to a cloud has become a critical concern for IT departments. In order to lower the barrier of pre-migration assessment, we have developed PseudoApp, a technique that can accurately predict the performance of applications in a cloud without actually installing the real application in the cloud.

The main idea of PseudoApp is to create a set of lightweight pseudo-application components that, at the individual request level, faithfully mimic the resource consumption behaviors of the real application's components. Our profiling tool collects the system call trace of the real application to automatically construct the corresponding pseudo-application. By benchmarking the simple pseudo-application in the cloud, we can accurately predict the performance of the real application.

Our current implementation of PseudoApp has several limitations. It does not accurately capture the instruction mix (e.g., integer, floating point, etc.) of the real application, and it does not accurately capture the real application's memory access pattern and the impact of L1/L2/L3 caches. These are major directions for future research.

References

[1] Amazon EC2. http://aws.amazon.com/ec2/.

[2] Average Web Page Size. http://www.websiteoptimization.com/speed/tweak/average-web-page/.

[3] K. Beyls and E. H. DHollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2001.

[4] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, Berkeley, CA, USA, 2003.

[5] force.com. The Force.com Multitenant Architecture: Understanding the Design of Salesforce.com's Internet Application Development Platform. In *Force.com Whitepaper http://wiki.developerforce.com/index.php/Multi_Tenant_Architecture*, 2008.

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

[7] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 114–122, New York, NY, USA, 2006. ACM.

[8] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 195–206, New York, NY, USA, 2006. ACM.

[9] LongJump. http://longjump.com/.

[10] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[11] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 37–48, New York, NY, USA, 2007. ACM.

[12] NYU TPC-W. http://www.cs.nyu.edu/pdsg/.

[13] L. P. Slothouber and P. D. A model of web server performance. In *In Proceedings of the 5th International World Wide Web Conference*, 1996.

[14] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A dollar from 15 cents: cross-platform management for internet services. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 199–212, Berkeley, CA, USA, 2008. USENIX Association.

[15] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, Berkeley, CA, USA, 2005. USENIX Association.

[16] 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 *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 19–19, Berkeley, CA, USA, 2009.

[17] V. Tran, J. Keung, A. Liu, and A. Fekete. Application migration to cloud: a taxonomy of critical factors. In *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, New York, NY, USA, 2011. ACM.

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

[19] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. Justrunit: experiment-based management of virtualized data centers. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, Berkeley, CA, USA, 2009. USENIX Association.