WebPerf: Evaluating "What-If" Scenarios for Cloud-hosted Web Applications

Yurong Jiang[†], Lenin Ravindranath[‡], Suman Nath[‡], Ramesh Govindan[†]
[†]University of Southern California [‡]Microsoft Research
[†]{yurongji,ramesh}@usc.edu [‡]{lenin, sumann}@microsoft.com

Abstract— Developers deploying web applications in the cloud often need to determine how changes to service tiers or runtime load may affect user-perceived page load time. We devise and evaluate a systematic methodology for exploring such "what-if" questions when a web application is deployed. Given a website, a web request, and "what-if" scenario, with a hypothetical configuration and runtime conditions, our methodology, embedded in a system called WebPerf, can estimate a distribution of cloud latency of the request under the "what-if" scenario. In achieving this goal, WebPerf makes three contributions: (1) automated instrumentation of websites written in an increasingly popular task asynchronous paradigm, to capture causal dependencies of various computation and asynchronous I/O calls; (2) an algorithm to use the call dependencies, together with online- and offline-profiled models of various I/O calls to estimate a distribution of end-to-end latency of the request; and (3) an algorithm to find the optimal measurements to take within a limited time to minimize modeling errors. We have implemented WebPerf for Microsoft Azure. In experiments with six real websites and six scenarios, the WebPerf's median estimation error is within 7% in all experiments.

CCS Concepts

•**Networks** → *Network performance analysis;* •**Software and its engineering** → *Application specific development environments;*

Keywords

Instrumentation; Async-Await; Dependency; What-if

1. INTRODUCTION

Many popular web applications have complex cloud architectures, with multiple tiers and inter-related components such as compute VMs, SQL and NoSQL storage, file systems, communication queues, third party services, geo-replicated resources, load balancers, *etc.* It is not uncommon for popular web applications to include tens of different types of off-the-shelf cloud resources [21, 43]. For example, `airbnb.com`, `alibaba.com`, and `netflix.com` use six different types of storage components [43].

Cloud providers such as Microsoft Azure and Amazon Web Services offer developers a large number of configuration choices in terms of alternative resource tiers, flexible geolocation, redundancy, *etc.* The choices come at varying cost and performance. Microsoft Azure offers 10 tiers of Web Servers (in June 2016), with the highest tier up to several orders of magnitude faster, but two orders of magnitude costlier, than the lowest tier (Table 1) for a website and various workloads we evaluate in §6.

While the choices give developers flexibility, choosing a configuration that optimizes cost and performance is challenging for various reasons (§2). Developers must carefully evaluate the application's performance under various configurations. They could try to do this based on cloud provider's SLAs, but these are often qualitative and insufficient to predict quantitative performance. Even if quantitative performance of a specific cloud resource under a configuration were available, they may need to translate that to end-to-end latency, which would require them to understand application logic and underlying concurrency. Developers could also try using data-driven techniques that rely on the application's historical performance on various configurations [10, 42]; but these can be slow and expensive, and are not suitable for first-time deployment.

In this paper, we address this challenge through the design, implementation, and evaluation of WebPerf, a tool that *estimates* the *distribution of cloud-side latency* of a web request under hypothetical changes (called what-if scenarios) to the *cloud-side configuration* of a web application. By repeatedly trying several what-if scenarios, a developer can quickly determine the best resource tier configuration that fits her budget.

The key idea behind WebPerf is to combine data-driven, offline latency models of various cloud APIs with application logic abstracted as causal dependency (*i.e.,* execution order) of various compute and I/O calls to compute the total cloud latency of a given request. This idea is motivated by

SIGCOMM '16, August 22-26, 2016, Florianopolis , Brazil
© 2016 ACM. ISBN 978-1-4503-4193-6/16/08. . . $15.00
DOI: http://dx.doi.org/10.1145/2934872.2934882

two key insights. First, causal dependencies of compute and I/O calls for a request remain relatively stable for the large class of what-if scenarios that WebPerf supports. Thus, it is possible to determine the dependencies once, and to reuse it for repeated what-if evaluations. Second, latencies of I/O calls to many cloud resources can be reliably modeled as functions of workload parameters such as storage table size and configuration parameters such as geolocation and resource tier, but independent of application logic. Therefore, WebPerf can build application-independent models of various cloud APIs *offline*, as a function of workload and configuration parameters, and use these models during what-if estimation.

Implementing the above idea for cloud-based web applications raises several challenges, which we address. First, while prior work has explored techniques to track causal dependencies in asynchronous applications [32, 33], modern web applications are written using task asynchronous paradigm [2] that prior work has not considered. We develop techniques to track causalities in such applications.

Second, predicting overall cloud latency by combining causal dependencies with latency models of cloud APIs is challenging for various reasons. Dependencies among compute and I/O calls in real applications can be complex. Moreover, latency models of cloud APIs are best represented as a distribution rather than as fixed values, and hence the prediction algorithm needs to combine the distributions. WebPerf uses a novel algorithm that hierarchically convolves these distributions, in a manner guided by the causal dependence. Finally, there are many other practical considerations that are needed for accurate prediction. A request may have nondeterministic causal dependency (*e.g.,* due to CDN hit or miss). Cloud resources can impose concurrency limits and hence introduce queueing delays. Latencies of some data-centric APIs (*e.g.,* to a SQL database) may depend on application and workload properties. WebPerf's prediction algorithm uses several techniques to handle such cases.

While WebPerf's primary goal is to predict an application's cloud-side latency, its prediction can be combined with network latency and client-side latency (predicted by existing tools such as WebProphet [25]) to produce an end-to-end predicted latency distribution. WebPerf proposes a Monte Carlo simulation-based algorithm to achieve this.

Finally, WebPerf must be fast so that a developer can quickly explore various what-if scenarios. Given a what-if scenario, WebPerf needs to build latency models for all computation and I/O calls in the application to reason about its baseline performance. To operate within a fixed measurement time limit, WebPerf formulates the measurement problem as an integer programming problem that decides how many measurements of various requests should be taken to optimize total modeling error (the process is similar to optimal experiments design [31, 44]). Our experiments show this can minimize modeling errors by $5\times$ on average.

We have implemented WebPerf for Microsoft Azure and evaluated it on six websites and six what-if scenarios. WebPerf's median estimation error was within $< 7\%$ in all cases.

2. BACKGROUND AND MOTIVATION

Platform as a Service (PaaS) cloud providers such as Amazon AWS and Microsoft Azure let developers rapidly build, deploy, and manage powerful websites and web applications. A typical cloud application has multiple tiers. (See [21, 43] for example architectures of real-world applications.) The back-end data tier consists of various data storage services, such as SQL databases, key-value stores, blob stores, Redis caches [34], *etc.* The front-end contains the core application logic for processing client requests. It may also include various cloud resources such as VMs, communication queues, analytics services, authentication services, *etc.* The front-end of a web application also includes a web server that generates HTML webpages to be rendered on browsers.

2.1 The need for what-if analysis

Cloud providers offer multiple resources for computation, storage, analytics, networking, management, *etc.* For example, Microsoft Azure offers 48 resource types in 10 different categories.[1] Each resource is usually offered in multiple *tiers* at different price, performance, and isolation level—for a Web server alone, Azure offers 30 different resource tiers (a few are shown in Table 1). Finally, cloud providers also permit redundancy and geolocation of resources. A single website may use multiple resources; for example, `airbnb.com`, `alibaba.com`, and `netflix.com` each use six different types of storage components [43]. Thus, a developer is faced with combinatorially many *resource configuration* choices in terms of the number of resource tiers, the degree of performance isolation across these tiers, the redundancy of resources, and geolocation of resources.

In this paper, we are primarily concerned with *cloud latency*, the time a user request spends in the cloud. Some of this latency is due to the network (when the front-end and back-end are not geographically co-located) and some of this latency is due to compute and storage access. Not surprisingly, the choice of a web application's cloud configuration can significantly impact cloud latency. A developer needs to be able to efficiently *search* the space of configurations to select a configuration that satisfies the developer's goal such as minimizing cloud latency of a request given a budget or meeting a deadline while minimizing the cost.

One approach could be to deploy each configuration and measure cloud latency; but this can be expensive and slow. Alternatively, the developer could try to determine cloud latency of a request from cloud providers' SLAs. This is hard because resource tier SLAs are often described qualitatively. Microsoft Azure lists performance of Web Server tiers in terms of CPU core counts, memory sizes, and disk sizes (Table 1). Redis cache performance is specified as *high*, *medium* or *low*. Translating such qualitative SLAs to quantitative performance is hard. Performance can also depend on runtime conditions such as load. Our results in Table 1 show that under no load, the lowest and the highest tier of Web Servers, whose prices differ by $100\times$, have relatively simi-

[1]All Azure offerings are reported as of June 2016.

Web Server Tier	Price (USD/month)	Configuration	Avg. Response Time/Request (ms)		
			1 req.	100 concurrent reqs.	90% CPU
Standard A0	15	1 Core, 0.75 GB Memory, 20 GB Disk	107.51	123837.6	283.6
Standard A1	67	1 Core, 1.75 GB Memory, 70 GB Disk	99.6	15770.8	210.3
Standard D2	208	2 Cores, 7 GB Memory, 100 GB Disk	97.4	2371.0	190.4
Standard A3	268	4 Cores, 7 GB Memory, 285 GB Disk	93.4	1720.70	142
Standard D12	485	4 Cores, 28 GB Memory, 200 GB Disk	96.2	1275.52	130.4
Standard D14	1571	16 Cores, 112GB Memory, 800GB Disk	90.1	752	115

Table 1—*A few tiers of Microsoft Azure Web Server: Price, Configuration, and Performance under various conditions. Response times are measured from a client in California to the index page of the SocialForum website (§ 6), deployed at a Web Server in US West.*

lar latency; but with 100 concurrent clients, the lowest tier is $164\times$ slower than the highest tier. Finally, even if it were possible to accurately quantify latency of a specific resource tier, estimating its impact on the total cloud latency requires understanding how I/O calls to the resource interleave with other I/O calls and how they affect the critical path of the web request. This is nontrivial.

To efficiently search the space of resource configurations, our paper develops WebPerf, a prediction framework that can perform *what-if analyses*—given a hypothetical resource configuration of an application and its workload, it can accurately predict a request's cloud latency without actually executing the application under the new configuration, and without relying on qualitative SLA descriptions, while still capturing interleavings between different I/O calls within a given request.

2.2 Key Insights

Unlike common website optimization tools [17, 45] that focus on webpage optimization, WebPerf enables developers to find low cloud latency resource configurations. The cloud latency of an application depends on (1) the causal order (*i.e.,* sequential or parallel) in which various computation and I/O calls happen, and (2) the latency of each computation and I/O call. We use the following two key insights to measure these two components.

► **Stable Dependency.** *Causal dependencies of various computation and I/O calls of a request in an application remain stable* over various what-if scenarios that we consider (Table 2). For example, if a request accesses a key-value store followed by accessing a blob store, it accesses them in the same causal order even when the key-value store or the blob store is upgraded to a different tier. Such determinism allows WebPerf to compute causal dependency for a request once and reuse it for repeated what-if analysis.

Of course, there can be nondeterminism in control paths—*e.g.,* a request may or may not query a database depending on whether a value is present in the cache. In that case, WebPerf issues the request repeatedly to stress various control paths and produces one estimate for each unique causal dependency. Such non-determinism, however, is relatively infrequent—in six real applications we evaluate in §6, only 10% requests demonstrate such nondeterminism. In §4.3, we describe how WebPerf handles variable latencies due to different control paths *inside* a cloud resource.

► **Application-independent API latency.** *The performance of individual I/O calls to* many *cloud resources can be reliably modelled independent of application logic, but as functions of various workload and configuration parameters.* For example, latency of a Redis cache lookup API does not depend on the application, but on its configuration such as its geolocation and its resource tier. Similarly, latency to a NoSQL table query API does not depend on application logic, but rather on workload parameters such as query type (*e.g.,* lookup *vs.* scan) and table size. Therefore, it is possible to build application-independent statistical models (called *profiles*) of these APIs offline, as functions of relevant workload and configuration parameters, and combine it with application-specific dependency information to estimate a request's cloud latency.

To verify application-independence, we measured the latency distribution of various I/O calls made by several real-world applications, and compared them with those generated by an application-independent profiler. Given an API and an application, we compute *relative error* of the profiler as $|l-l'|/l$, where l and l' are the latencies of the API measured from the application and the profiler respectively. Figure 1 shows mean and 90th percentile relative errors for all 11 I/O APIs used by SocialForum, a real Azure application we describe in §6. The mean error ranges from 0.4% to 6.5% while the 90th percentile error ranges from 5% to 10%. Overall, the error is relatively small for other scenarios, confirming our hypothesis. We observed similar results for four other applications we evaluate in §6, as well as for two popular Wordpress plugins for Azure blob storage and Redis cache.

Latency profiles may depend on configuration and workload parameters. While required configuration parameters come directly from a given what-if scenario, workload parameters need to come from developers and applications. WebPerf uses several techniques, including application-specific *baseline latencies* and developer-specified *workload hints*. We describe these in §4.3 and evaluate them in §6.

2.3 WebPerf Overview

WebPerf is designed to satisfy three requirements: it must estimate the cloud latency distribution of a given request and a what-if scenario *accurately*, must do so *quickly* so that developers can explore many scenarios in a short time, and must require *minimal developer input*.

Figure 3 depicts various components of WebPerf. It takes as input a workload and a what-if scenario. The workload consists of (1) a web application, (2) a set of HTTP requests, and (3) optional *workload hints* to help WebPerf choose accurate latency models for the application's workload. Table 2 lists various what-if scenarios WebPerf supports.

WebPerf works as follows. WebPerf's Binary Instru-

1	Delete***(Async)*** (T) ★	5	ExecuteQuerySegmented (T)	9	ToList (S)
2	UploadFromStream (B)	6	SortedSetRangeByValue (R)	10	Send (R)
3	AddMessage(Q)	7	StringGet (R)	11	ReadAsString (B)
4	Execute (T)	8	SaveChanges (S)		

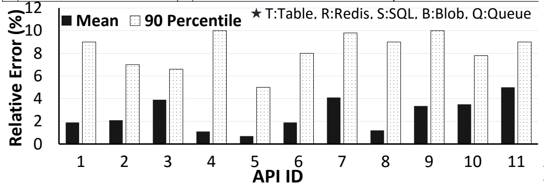


Figure 1— *Relative errors of app-independent profiles.*

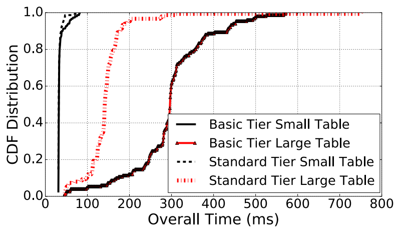


Figure 2—*Latency of a Join in Basic and Standard tier of Azure SQL.*

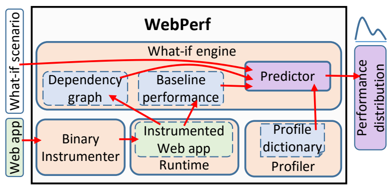


Figure 3— *WebPerf Architecture.*

What-if scenario	Example
Location: A resource X is deployed at location Y	X = A Redis Cache or a front end, Y = Singapore
Tier: A resource X is upgraded to tier Y	X = A Redis cache, Y = a standard tier (from a basic tier)
Input size: I/O calls to resource X take inputs of size Y	X = a blob storage, Y = a 760×1024 image
Load: X concurrent requests to resource Y	$X = 100$, Y = the application or a SQL database
Interference: CPU and/or memory pressure, from collocated applications, of $X\%$	X = 50% CPU, 80% memory
Failure: An instance of a replicated resource X fails	X = A replicated front-end or SQL database

Table 2— *What-if scenarios supported by WebPerf*

menter (§3) automatically instruments the given web application. The requests in the workload are then executed on the instrumented application, generating two pieces of information: (1) a *dependency graph*, showing causal dependencies of various compute and I/O calls executed by the requests, and (2) the *baseline latency* of various compute and I/O calls for the initial configuration. The number of measurements collected for the baseline latency is decided by an optimization algorithm (§5.2). The instrumentation is disabled after the measurement finishes. WebPerf also uses a Profiler (§4), which builds, offline, empirical latency distributions of various cloud APIs under various what-if scenarios. The core of WebPerf is the What-If Engine (§4), which combines dependency graphs, baseline latencies, and offline profiles to predict a distribution of cloud latencies of given requests under what-if scenarios.

WebPerf can also combine its predicted cloud latencies with network latencies and client-side latencies (*e.g.,* predicted using WebProphet [25]) to produce a distribution of end-to-end latencies (§5.1).

3. TRACKING CAUSALITY IN TASK ASYNCHRONOUS APPLICATIONS

WebPerf's prediction algorithm uses causal dependencies between various computation and I/O calls invoked by a request. Cloud applications are increasingly written using *Task Asynchronous Paradigm*, which prior work (§7) on causal dependency tracking does not consider. WebPerf can track causal dependency within such applications *accurately*, *in a lightweight manner*, and *with zero developer effort*.

```
async Task<Response> ProcessReq(Request req) {
    /* Synchronous process block 1 */
    var input = GetInput(req);
    var task1 = AsyncTask1(input);
    var output = await task1;
    /* Synchronous process block 2 */
    var response = GetResponse(output);
    return response;
}
async Task<Output> AsyncTask1(Input input) { ... }
```

Figure 4— *Async-await example.*

3.1 Task Asynchronous Programming

Node.js [28] popularized the notion of non-blocking I/O in cloud and server applications. Non-blocking APIs greatly increase an application's throughput in a lightweight manner (*i.e.,* with a small thread pool). Today all major programming languages support non-blocking operations.

There are two major flavours of non-blocking I/O. The *Asynchronous Programming Model* (APM) uses callbacks that are invoked on certain events such as completion of an operation. Many languages including JavaScript, Java, and C# support APM. A major drawback of APM is that developers often need to write a large number of callback methods to sequence their request pipeline, quickly getting into the problem of *callback hell* [7]. Moreover, callbacks invert the control flow, and can obfuscate developer's intent [41].

These limitations are addressed by the *Task Asynchronous Paradigm* (TAP), which allows writing non-blocking asynchronous programs using a syntax resembling synchronous programs (Figure 4). TAP is supported by many major languages including .NET languages (C#, F#, VB), Java, Python, JavaScript, and Scala. For instance, C# supports TAP with a `Task` object and `async` and `await` constructs (example code later). TAP has become increasingly popular for writing cloud applications. Almost all Microsoft Azure framework libraries today support TAP as the *only* mechanism for doing asynchronous I/O and processing. Amazon AWS also provides TAP APIs for .NET [3].

Figure 4 shows an example request pipeline written with TAP in C#. `ProcessReq` method processes the incoming request, does a non-blocking asynchronous call (`AsyncTask1`) to, say, fetch a document from the store, processes the result, and sends a response back. `AsyncTask1` is an *async* method that starts the task asynchronously and returns a `Task` object. To obtain a result for

```
/* Receive request */
var task1 = AsyncTask1(...);
var task2 = AsyncTask2(...);
var result = await Task.WhenAll(task1, task2);
/* Send response */ ... /* Receive request */
var task1 = AsyncTask1(...);
var task2 = AsyncTask2(...);
var result = await Task.WhenAny(task1, task2);
/* Send response */
```

Figure 5— *When all and when any example.*

an *async* method, the caller does *await* on the task object. When *await* is used, the execution does not proceed until the result of the awaiting task is obtained. Note that *await* does not block the thread, but returns the thread and adds the rest of the method as a *continuation* (implicit callback) to the task being awaited. In Figure 4, the continuation is the start of the `synchronous block 2`. When the task finishes, its continuation executes on a different logical thread. The execution trace for `ProcessReq` is shown in Figure 6.

Synchronization Points. TAP provides explicit abstractions for developers to synchronize multiple parallel tasks. .NET provides two such abstractions: `Task.WhenAll` and `Task.WhenAny`. WhenAll accepts multiple tasks as argument and signals completion only when *all* tasks complete. Figure 5 (top) and Figure 7 (left) show an example request pipeline and its execution trace where the response is returned only when both tasks finish. WhenAny accepts multiple tasks but signals completion as soon as *any one* of them completes. Figure 5 (bottom) and Figure 7 (right) show an example request pipeline and its execution trace. WhenAll and WhenAny return tasks that can be awaited or synchronized with other tasks.

3.2 Tracking Causal Dependency

Given a user request in a TAP application, we would like to extract a causal dependency graph consisting of three pieces of information: (1) *nodes* representing synchronous and asynchronous compute and I/O calls, (2) *edges* representing causal (*i.e.,* happens-before) dependency among nodes, and (3) WhenAll and WhenAny *synchronization points*. All these pieces are required for total cloud latency estimation. Missing nodes may result in latency underestimation. Missing or incorrect edges may imply incorrect execution orders (serial *vs.* parallel) of nodes and produce inaccurate estimates.

Existing causality tracking techniques, however, may not capture all these necessary pieces of information (§7). A network-level proxy can externally observe I/O calls and their timing information [10], but will miss computation nodes and synchronization points that are not observable at the network layer. Moreover, inferring dependency based on timing information can be wrong. Suppose tasks n_1 and n_2 run in parallel and n_3 starts only after n_2 finishes. Each node has an execution time of t, and hence the total execution time is $2t$. Externally observed timing and execution order could infer that n_3 is dependent on *both* n_1 and n_2. This can lead to incorrect estimation in a what-if scenario that considers the case when n_1's execution time is $t/2$. The correct dependency graph will estimate a total execution time of $1.5t$, while the externally observed dependency graph could estimate $2t$. One might be able to correct some of these ambiguities by collecting a large amount of data (as in [10]). WebPerf, however, is designed to be used at deployment time when not much data is available.

Another way to track causal dependency is to instrument the runtime. Existing .NET profiling tools [40, 22, 1] can capture some dependencies, but fail to capture asynchronous dependencies and synchronization points of async-await programs. AppInsight [32] can track asynchronous callback dependencies, but it is designed for APM and does not support TAP.

The amount of information missed by existing solutions (and hence the estimation error) can be significant. In workloads from six real applications we describe in §6, a dependency graph on average has 182 computation nodes (missed by a network proxy), 180 async-await edges (missed by all), and 62 synchronization points (missed by all).

3.3 Capturing an Execution Trace

We now describe how WebPerf captures all the information in dependency graphs for TAP. It uses a two-step process: it instruments the application binary to capture the execution trace of a request, and then analyzes the execution trace to construct the request's dependency graph.

WebPerf automatically instruments application binaries to capture the execution trace that preserves causal dependencies of all async calls. Doing this for a TAP application presents several unique challenges not addressed by existing causality tracking solutions (*e.g.,* AppInsight) for APM applications. We now describe the key challenges and our solutions. The techniques are described for .NET, but the general idea can be used for other languages supporting TAP.

Tracking implicit callbacks: To track the lifetime of an asynchronous call, we need to correlate the start of the call and its callback. In APM, callbacks are explicitly written by developers and these callbacks can be instrumented with a correlation ID to match with the original request. In TAP, however, there are no such explicit callbacks – upon await call on a (completed) task, execution starts from its *continuation* (*i.e.,* rest of the async method after the await call). Thus, continuations act as implicit callbacks.

Identifying and instrumenting continuations in the source code may not be obvious; for instance, a task's creation and continuation can be in two different methods. To address this, we observe that an async method executes as a state machine, which maps continuations to different states, and state transitions occur as awaited tasks complete. In fact, .NET compiles async-await source code to binaries containing state machines. For instance, consider the async method in Figure 4. The .NET compiler translates the code to a state machine containing two key states: the first state contains the synchronous block 1 and starts the await process, and the second state contains the continuation of the await (*i.e.,* synchronous block 2). The state machine also contains a `MoveNext()` method that causes transition between states when awaited tasks complete.

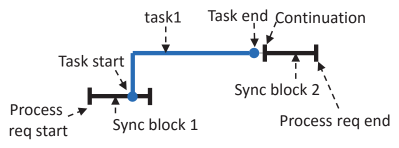


Figure 6— *Execution trace for the code shown in Figure 4.*

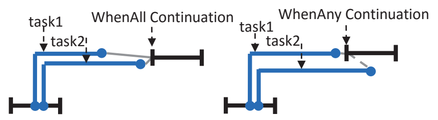


Figure 7— *Execution traces for the code shown in Figure 5.*

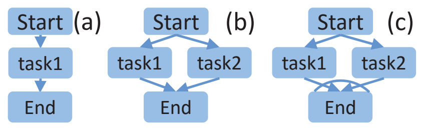


Figure 8— *Dependency graphs for the requests in Figures 4 and 5.*

```
1  class ProcessReq__ {
2      Request req;
3      int state = −1;
4      TaskAwaiter awaiter;
5      AsyncTaskMethodBuilder builder;
6      int asyncId = -1;
7      public void MoveNext() {
8          asyncId = asyncId == -1 ? Tracker.AsyncStart() :
               asyncId;
9          Tracker.StateStart(asyncId);
10         switch (state) {
11             case −1:
12               state = 0;
13               /∗ Synchronous process block 1 ∗/
14               var input = GetInput(req);
15               var task1 = AsyncTask1(input);
16               Tracker.TaskStart(task1, asyncId);
17               awaiter = task1.GetAwaiter();
18               builder.OnCompleted(awaiter, this);
19               Tracker.Await(task1, asyncId);
20               break;
21             case 0:
22               /∗ Synchronous process block 2 ∗/
23               var output = awaiter.GetResult();
24               var response = GetResponse(output);
25               builder.SetResult(response);
26               Tracker.StateEnd(asyncId);
27               Tracker.AsyncEnd(asyncId);
28               return;
29         }
30         Tracker.StateEnd(asyncId);
31     }
32 }
```

Figure 9— *Async state machine for the code in Figure 4. Statements in bold show the instrumented code. For clarity, we show the C# code instead of bytecode.*

Since the underlying state machine separates continuations as explicit states, WebPerf instruments the state machine (at bytecode level) rather than the original code. This is in contrast with AppInsight, which can instrument source code or equivalent binaries to track APM dependencies.

Given a state machine of an async method, WebPerf instruments its `MoveNext()` method (in Figure 9, statements in bold show the instrumentation code). **1)** When an async method starts, we generate a new async id and save it as part of the state machine (line 8). **2)** For async tasks (I/O, large compute), we monitor the start of the task (line 16) connecting it to the async method using the async id. **3)** We connect task completion to its continuation by tracking task awaits (line 19). We use the task object id, thread id and async id to accurately connect them – this ensures correctness even if task start and task await are part of different methods and irrespective of the task completion time. **4)** We track the lifetime of a thread by tracking the start and end of a state in the state machine (line 9, line 20, and line 30). **5)** Finally, we track the completion of an async method by tracking calls to `SetResult`, an inbuilt method that triggers the response of an async method (line 25).

Tracking pull-based continuation: Execution of a continuation is pull-based – it is invoked only when await is called and this can happen much later than the actual task completion. Hence, we need to track task completions independently from awaits but also understand their causal dependencies. In contrast, in APM, the framework invokes the callback as soon the task completes. To accurately track task completion, we add a continuation to the task (inside the `Tracker.TaskStart` method in line 16) and wrap the completion handler inside an object that saves the async id.

Tracking execution forks and joins: To track synchronization dependencies between tasks, we instrument the code to intercept `Task.WhenAll`, `Task.WhenAny` and other equivalent primitives. Using object ids of tasks passed to these calls, we record their dependencies. When tasks complete, we connect the continuations to the set of parallel tasks that were synchronized.

Tracking non-TAP calls: Apart from async-await calls, WebPerf also tracks other types of asynchronous calls (*e.g.,* APM with explicit callbacks) and expensive synchronous calls (*e.g.,* synchronous Azure APIs). For the former, WebPerf uses similar instrumentation technique as AppInsight. For the latter, it logs start and end of the call and uses thread ids to track their dependencies with async methods.

Optimizations: We use a few optimizations to reduce the runtime overhead of instrumentation. First, we track only (synchronous and asynchronous) APIs having known method signatures from Azure libraries. Second, within an async-await method call chain, we instrument only the non-blocking tasks (with known signatures). In .NET, methods using *await* should also be declared as *async* and can be awaited by other methods. Hence, when a request involves a non-blocking async call, all methods in its pipeline including the entry point are declared as async methods and compiled into a state machine. Though the intermediate methods are async, continuations between them are called synchronously except for the method with the non-blocking call. We significantly reduce the overhead of tracking without compromising accuracy by monitoring only non-blocking async tasks, state machines of methods awaiting them, and the request entry point method. Our instrumentation overhead is low and comparable to numbers reported in AppInsight [32, 33].

3.4 Extracting dependency graphs

The dependency graph of a request is a directed acyclic graph with three types of nodes: *Start*, *End*, and *Task*. *Start* and *End* denote the start and the end of the request. *Task* nodes represent async API calls, as well as other synchronous and asynchronous compute and I/O calls that we want to profile and predict. An edge $\mathbf{A} \rightarrow \mathbf{B}$ means that **B** can start only after **A** finishes. Multiple tasks originating from a single task indicate parallel tasks. Multiple tasks terminating at a task indicates a `WhenAll` or `WhenAny` dependency.

WebPerf processes the execution trace from start to end to construct a dependency graph. It constructs a *Task* node for each unique async call, compute or I/O call with known signatures, and expensive computation. It constructs an edge $t \rightarrow t'$ when it encounters in the execution trace a task t' starting in the continuation of another task t. Note that the same continuation thread of a task t may contain start of multiple tasks $t_1, t_2, \cdots$, resulting in parallel edges $t \rightarrow t_1, t \rightarrow t_2, \cdots$. On encountering `WhenAll` or `WhenAny` method call with tasks $(t_1, t_2, \cdots)$ as input Task arguments, WebPerf constructs a synchronization *Task* t representing the continuation of the method, and edges $t_1 \rightarrow t, t_2 \rightarrow t, \cdots$ representing synchronization dependencies. Synchronization tasks also contain information about whether the dependency is `WhenAll` or `WhenAny`, the information is show as an arc over all incoming edges (or no arc) for `WhenAny` (or `WhenAll` respectively).

Figure 8a–c shows dependency graphs for the execution traces in Figure 6, 7 (left), and 7 (right) respectively. We put an *arc* over incoming edges for `WhenAny` dependency, to differentiate it from `WhenAll` dependency. Note that the execution trace also contains timings for compute threads (black horizontal lines); hence, WebPerf can profile large compute components, in addition to synchronous and asynchronous tasks, without knowing their semantics.

4. EVALUATING WHAT-IF SCENARIOS

WebPerf estimates the cloud latency of a request under a what-if scenario in three steps: (1) building application-independent and parameterized latency profiles of various APIs offline; (2) computing the dependency graph of a given request (§3) and application-specific baseline latency distributions of various tasks in the request; (3) predicting the cloud latency by combining the dependency graph, baseline latencies, and latency profiles for the given what-if scenario.

4.1 Application-independent Profiling

WebPerf maintains a *profile dictionary*, containing *profiles* or statistical models of latencies of different cloud APIs under various configurations and inputs. In our implementation, profiles are modeled by nonparametric latency distributions and are stored as histograms. WebPerf uses two types of profiles: *independent* and *parameterized*, differentiated by whether they depend on workload parameters or not.

Workload-independent profiles. Independent profiles model APIs whose performance does not depend on applications or workloads, but may depend on configurations.

To profile an API for a specific cloud resource R (*e.g.,* a Redis cache), WebPerf repeatedly calls the API until it gets a *good enough* latency distribution (*i.e.,* when the sample mean is within two standard errors of the true mean with 95% confidence level [37]). For all Microsoft Azure APIs, WebPerf needs fewer than 100 measurements. To support various what-if scenarios, WebPerf's profiler builds profiles for each API under different configurations of R, input parameters of the API, system loads, *etc.*, as well as at different times incrementally to capture temporal variabilities. Table 2 shows various what-if scenarios WebPerf currently supports. To build profiles for location, tier, input size, and load, the profiler deploys R at different locations or tiers, and issues requests to them with different input sizes and concurrent loads, respectively. For the CPU interference scenario, WebPerf builds profiles for client side CPU processing overhead of calling the APIs. It also empirically builds a mapping between CPU time and wall clock time under different background CPU stresses, which it uses to convert the CPU time profiles to wall clock time profiles. To profile replica failure scenarios, WebPerf computes expected increase in loads on working instances of R and uses the load profiles to approximate failure profiles.

Parameterized profiles. Performance of a small number of cloud APIs depends on workloads. For instance, the query API to Azure SQL exhibits different latencies based on the specific query (*e.g.,* whether it has a join) and table size. A cloud resource can also exhibit variable latencies due to different control paths based on an application's workload (*e.g.,* CDN latency for cache hit vs cache miss). WebPerf builds multiple profiles for each of these APIs, parameterized by relevant workload parameters. WebPerf allows developers to provide *workload hints*, based on which it chooses the right profile for an API. WebPerf exposes to developers a list of APIs for which workload hints can be provided.

For example, WebPerf profiles Azure CDN's latency as two distributions, one for cache hits and one for misses, and allows developers to specify a target cache hit rate as a workload hint. WebPerf then appropriately samples from the two profiles to generate a (bimodal) distribution of latencies under the specified hit rate. §6.6 shows an example. For the query API to Azure Table storage and SQL, WebPerf builds multiple profiles, one for each table size (*e.g.,* $< 1MB$, $< 10MB$, *etc.*) and query complexity (*e.g.,* lookup, scan, or join). The developer can provide workload hints in terms of table size and query complexity, and WebPerf chooses appropriate profiles based on those workload parameters.[2]

Note that building profiles for all APIs under all hypothetical configurations and workload scenarios can be very expensive and slow. WebPerf therefore seeds its profile dictionary with profiles for only a few popular APIs under a few common what-if scenarios and workload parameters; it

[2]WebPerf's SQL profiles are only crude approximations. SQL query performance can depend on factors such as indexing, query optimization, *etc.* that are beyond the scope of the paper. Therefore, WebPerf's predictions for scenarios involving SQL are not as accurate as those for other scenarios.

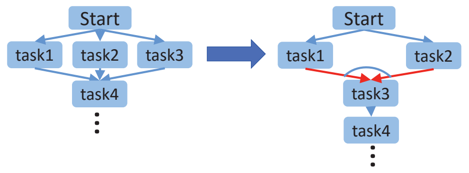


Figure 10—*Imposing Concurrency Limits.*

builds the remaining profiles only on demand, in parallel, and reuses these in future what-if analysis. Note that, only a small number of APIs in Azure exhibit workload-dependent performance, and hence need to be parameterized. Building parameterized profiles for these small number of APIs is tractable in practice.

4.2 Application-specific Profiling

In this step, WebPerf executes a given request on an instrumented (§3) version of the application to collect its dependency graph. It also executes the request multiple times to capture an application-specific, *baseline latency* distribution of each task of the graph. Again, enough samples are collected to obtain a *good enough* latency distribution for each task. Baseline profiles capture any application-specific factor that might affect API latencies and hence, in principle, are more accurate than offline profiles. Therefore, WebPerf prefers, whenever possible, baseline latencies over offline latencies for prediction.

Measuring baseline latencies is in the critical path of what-if analysis; WebPerf carefully decides how many measurements to take for each request within a given time budget, as described in §5.2.

4.3 What-if Engine

The What-If Engine of WebPerf takes the following as inputs: (1) profiles of various APIs, (2) the dependency graph and the baseline latency of tasks, (3) a what-if scenario, such as the one from Table 2. It produces a distribution of estimated *cloud latency*. The basic idea is as follows: given the dependency graph, WebPerf first estimates the latency distribution of each task node, and then estimates the distribution of total latency of the request by combining the per-task latency distributions. The overall process has multiple steps and challenges; we now describe these challenges and how we address them.

Profile selection. WebPerf uses the following workflow to select an appropriate profile for each node in the dependency graph. (1) If the node (*i.e.,* the API called in the node) is not affected by the what-if scenario, WebPerf uses the API's baseline profile as its estimated latency distribution. (2) If the node is affected by the what-if scenario, WebPerf checks whether the API has a parameterized profile and the developer has provided appropriate workload hints. If so, WebPerf chooses the correct parameterized profile as its estimated latency. Otherwise, it uses the API's independent profile as the estimated latency.

For instance, consider a request involving API calls to a Redis cache and a SQL database. For a what-if scenario for upgrading the Redis tier, WebPerf uses the application-independent profiles of the target tier of Redis APIs (since Redis performance prediction does not need developer hints) and baseline profiles of the SQL APIs (since they are not affected by the what-if scenario).

Algorithm 1 CLOUD LATENCY PREDICTION

```
 1: INPUT: A node $N$ in a dependency graph, Baseline latency
    distribution $\mathcal{B}$, Profile dictionary $\mathcal{P}$, a what-if scenario $\mathcal{S}$)
 2: Function CloudLatencyPredict $(N, \mathcal{B}, \mathcal{P}, \mathcal{S})$
 3: $L_{any} \leftarrow \{\}$
 4: $L_{all} \leftarrow \{\}$
 5: for every parent node $N'$ of $N$ do
 6:     if $N'$ appears in $\mathcal{S}$ then
 7:         $\rho \leftarrow$ latency distribution of $N'$ from profile dictionary $\mathcal{P}$
 8:     else
 9:         $\rho \leftarrow$ latency distribution of $N'$ from baseline latency $\mathcal{B}$
10:     $D \leftarrow$ ProbAdd( $\rho$, CloudLatencyPredict $(N', \mathcal{B}, \mathcal{P}, \mathcal{S})$)
11:     if the edge $N' \rightarrow N$ is a $WaitAny$ edge then
12:         $L_{any} \leftarrow L_{any} \cup D$
13:     else
14:         $L_{all} \leftarrow L_{all} \cup D$
15: return ProbMax( ProbMin( $L_{any}$), ProbMax($L_{all}$))
OUTPUT: Predicted latency distribution of node $N$
```

Enforcing concurrency limits. Some cloud APIs have implicit or explicit concurrency limits that can change across tiers. For instance, Azure SQL supports maximum 30 and 60 concurrent connections in Basic and Standard tier respectively. Such limits impose queueing latency when the number of concurrent connections goes above the limit.

To account for such queueing latency, WebPerf modifies the given dependency graph to impose concurrency limits. For a known system-wide limit of maximum n concurrent calls to a specific I/O and for a load of r concurrent user requests, WebPerf imposes a limit of n/r concurrent calls per request. It then traverses the dependency graph of the request top down and checks if any task reaches the limit of n/r; if so, it removes the task (along with all its descendants) from its parent, and adds it as a *WhenAny* child of its in-limit siblings. The basic idea is shown in Figure 10. Given a system limit of two, WebPerf moves `task3` with all its descendants as a *WhenAny* child of `task1` and `task2`.

Probabilistic estimation. Finally, WebPerf combines (using Algorithm 1) estimated latency distributions of all tasks in the (modified) dependency graph to produce one distribution of total cloud latency. The algorithm works *bottom-up* and is invoked with the End node of the dependency graph. Given a node, it estimates its latency based on baseline latency (if the node is not affected by what-if scenario or if the node is application-dependent) or offline profile (otherwise). It then recursively computes latency distribution of each path to it and combines them based on the semantics of the synchronization points.

Algorithm 1 would have been simpler if nodes had scalar latencies. However, WebPerf models node latencies as probabilistic distributions and they need to be combined according to how various tasks execute—in sequence or in parallel, and whether execution waits for all tasks to finish or any one to finish. Consider tasks $t_i, 1 \leq i \leq k$ in the dependency graph. Suppose the latency distribution of task t_i

is represented by an i.i.d. random variable X_i with distribution function $f_i(x)$. Suppose Z is the random variable representing the total execution time of all tasks. Z can be estimated based on how the tasks execute and how the application waits for their execution to finish.

If all tasks run sequentially (Figure 8(a)), $Z = \sum_{i=1}^{k} X_i$. Then the distribution function $f(z)$ of Z is given by $f(z) = f_1(x) * f_2(x) * \dots f_k(x)$, where $*$ denotes the convolution operator. For $k = 2, f(z) = f_1(x) * f_2(x)$ and $\Pr(Z = z) = \sum_w \Pr(X_1 = w) \Pr(X_2 = z - w)$. Since convolution is associative and commutative, the computation can be extended to $k > 2$.

With parallel execution, there are three cases depending on whether the application needs to wait for *all* or *any* of the tasks to finish. **(1)** Wait for *all* to finish (Figure 8(b)): In this case $Z = \max_{i=1}^{k} X_i$. The distribution function for Z can be computed as follows: $\Pr(Z \le w) = \Pr(X_1 \le w, X_2 \le w, \dots, X_k \le w) = \prod_{1 \le i \le k} \Pr(X_i \le w)$. **(2)** Wait for *any* to finish (Figure 8(c)): In this case, $Z = \min_{i=1}^{k}(X_i)$ The distribution function of Z can be computed in a similar manner to the Max function. **(3)** A combination of the above two: Suppose the application waits for any of $X_1, \dots, X_m$ to finish, and all of $X_{m+1}, \dots, X_k$ to finish; then $Z = \max(\min(X_1, \dots, X_m), \max(X_{m+1}, \dots, X_k)$. The distribution function can be computed by combining the above distributions for Max and Min.

Algorithm 1 implements the computations using three building blocks that involve operations on discrete distributions: **ProbMax**, **ProbMin** and **ProbAdd**. **ProbMax** and **ProbMin** are computed based on **(1)** and **(2)** above. **ProbAdd** for adding two distributions is based on convolutions [18] (see associated technical report [23] for details). Given a node, Algorithm 1 recursively computes the latency distribution for every path to it (by applying **ProbAdd** on latency estimation of nodes on the path), and applies **ProbMax** or **ProbMin** to those paths, depending on whether the paths have $WhenAny$ edges or not.

5. WebPerf EXTENSIONS

This section describes two extensions to WebPerf. We have implemented them, and we evaluate them in §6.

5.1 End-to-end Latency Prediction

WebPerf's cloud latency estimates can be extended to estimate end-to-end latency of a web request. End-to-end latency consists of three components: (1) cloud-side latency given by the prediction algorithm above, (2) network latency between an application's cloud frontend and the client, and (3) client-side latency within client's browser, which can be predicted by tools such as WebProphet [25]. Accurately modeling network latency is outside the scope of the paper. For simplicity, we model network latency using the RTT distribution between the client's browser and the application frontend. WebPerf uses a combination of two techniques to estimate end-to-end latencies: *end-to-end tracking* and *probabilistic estimation*.

End-to-end tracking. To match a client side webpage request with corresponding HTTP requests seen by the cloud frontend, WebPerf automatically instruments the webpage so every HTTP request from it contains a unique id. WebPerf also instruments the HTTP module at the webserver to match the request with the cloud-side processing.

Algorithm 2 END-TO-END LATENCY PREDICTION

```
 1: INPUT: A webpage $W$ and a latency vector $\bar{L}$ of all objects in $W$)
 2: Function E2ELatencyPrediction ($W, \bar{L}$)
 3: $DG \leftarrow$ dependency graph of objects in $W$
 4: $S \leftarrow \{\}$
 5: while more sample to collect do
 6:     for every object $O$ in $W$ do
 7:         if $O$ involves web request from cloud frontend then
 8:             $\omega_{Net} \leftarrow$ a Monte Carlo sample from RTTDistribution
 9:             $\omega_{Cloud} \leftarrow$ a Monte Carlo sample from CloudLatencyPredict() for $O$
10:             $\omega \leftarrow \omega_{Cloud} + \omega_{Net}$
11:         else
12:             $\omega \leftarrow$ a Monte Carlo sample from FetchTime($O$)
13:         Update $O$'s latency in $\bar{L}$ with $\omega$
14:     $S \leftarrow S \cup$ ClientPrediction($DG, \bar{L}$)
15: return Distribution of $S$
OUTPUT: Predicted latency distribution of $W$
```

Probabilistic estimation. WebPerf uses Monte Carlo simulation to add cloud, network, and client latency distributions to produce an end-to-end latency distribution. It uses a WebProphet-like tool called **ClientPrediction** that, given a client-side dependency graph DG_{client} with causal dependencies of various objects downloaded by the browser and a latency vector $\bar{L}$ of download latency of all objects in the webpage, can estimate the page load time.

Algorithm 2 uses **ClientPrediction** for end-to-end prediction as follows. It performs the following steps until a sufficient number of end-to-end latency samples are collected (*e.g.,* until the confidence bound is within a target). For each object O in the webpage, if it comes from a frontend HTTP call, it samples a random value from the cloud-side latency distribution and a random value from the network RTT distribution and adds them to estimate the download time for O. On the other hand, if O comes from external sources or if it is a static object, its download time is sampled from a **FetchTime** distribution. (The **FetchTime** distribution is profiled offline. If no such distribution is available for an object, its value in $\bar{L}$ can be used as the sampled value.) The sampled download time is then updated in $\bar{L}$ and passed to **ClientPrediction** to produce a sample of end-to-end latency. Finally, the algorithm returns the distribution of all samples.

5.2 Optimal Profiling

WebPerf's primary bottleneck is estimating baseline latencies of I/O calls appearing in the target workload (*i.e.,* set of requests). Given a time budget T for measuring n requests, one straightforward way is to allocate T/n time for each request. However, this is not optimal since modeling some APIs would need more samples than modeling others, due to their different variabilities in latencies. WebPerf uses an algorithm to decide how to allocate the time budget optimally to measure different APIs. The process is similar to optimal experiments design [31], a statistical technique that allows us to select the most useful training data points, and has been used for performance prediction for large-scale advanced analytics [44].

Suppose the developer is interested in what-if analysis of her application for n different requests: $r_1, r_2, \ldots, r_n$. Let us first start with the simplified assumption that each request r_i contains exactly one I/O call c_i. We need to determine $n_i, 1 \leq i \leq n$, the number of times r_i should be measured to build a profile of its I/O call c_i. We have two goals. First, the total measurement time[3] for all requests must be within a given budget T. Suppose executing request r_i takes t_i seconds; then $\sum_{i=1}^{n} n_i t_i \leq T$. Second, the total (or average) standard error of measurements is minimized. Suppose executing request r_i takes t_i seconds on average, with a standard deviation of σ_i. Then, the standard error of n_i measurements of r_i is given by $\sigma_i / \sqrt{n_i}$. Thus, we want to minimize $\sum_{i=1}^{n} \sigma_i / \sqrt{n_i}$. In addition to the above two goals, we also want each request to be measured at least k times, in order to get meaningful statistics. The above problem can be formulated as the following integer program:

$$\min \sum_{1 \leq i \leq n} \frac{\sigma_i}{\sqrt{n_i}}$$

$$\text{s.t.} \sum_{1 \leq i \leq n} n_i t_i \leq T, n_i \geq k \qquad \forall i = 1, \ldots, n$$

The problem is in NP, since a simpler version of the problem can be reduced to the Knapsack problem. We propose a linear analytical approximation algorithm based on Lagrange multipliers [6] to solve it. We can derive the value for each $n_i = (T(\frac{\sigma_i}{2t_i})^{\frac{2}{3}})/(\sum_i (\frac{\sigma_i \sqrt{t_i}}{2})^{\frac{2}{3}})$, which is a real number larger than 0. The intuition behind this is that we need more samples for those APIs with higher variance but take less time. We approximate them by rounding to closest integers, which gives the optimal number of requests.

We have generalized our solution [23] to the more realistic setting where a request can contain multiple I/O calls and an I/O call can appear in multiple requests.

6. EVALUATION

We have implemented WebPerf for Microsoft Azure web apps, and we use this to evaluate WebPerf.

6.1 Methodology

Implementation. The WebPerf binary instrumenter is implemented as a NuGet package [29] that a developer can obtain from Azure Site Extension Gallery [4] and install to her web app as a site extension. The *Profiler* is implemented as a collection of web applications and the profile dictionary is stored in an Azure table. The WebPerf *What-if Engine* is implemented as a cloud service with a web interface that allows the developer to specify her web app, workload, and what-if scenarios. The engine communicates with the Instrumenter (included in the target web app by the developer) and the Profiler through HTTP requests. The WebPerf prediction component is written in python ($\sim$ 20K LOC).

[3]The problem can also be formulated with other costs.

Experimental Setup. We have evaluated WebPerf with six third party Azure web applications, listed in Table 3, for which we could find cloud-side source code or binaries. The key application for which we provide detailed results is **SocialForum**,[4] a production-ready Microsoft web application. It provides Instagram-like social network functionality and allows users to create new accounts, create/join forums, post/share/tag/search pictures and comments, *etc.* It uses five different Azure services: *Azure Blob Storage* for storing large data such as images; *Azure Table* for storing relational data; *Azure Redis Cache* for caching and storing key-value pairs; *Azure Service Bus* for queueing background processing tasks and *Azure Search Service* for searching forums. The index page of the SocialForum website consists of more than 20 objects, and the corresponding HTTP request at the cloud side has a dependency graph consisting of 116 async I/O calls to Redis Cache, Table storage, and Blob storage, with many executing in parallel.

For lack of space we omit the detailed architecture of remaining five applications in Table 3; these are discussed in [23]. They all are of modest complexity, as hinted by various Azure resources they use. The requests that we use to all six applications are also fairly complex, with large dependency graphs. On average, a dependency graph has 182 nodes, 180 async-await edges, and 62 synchronization points.

6.2 Cloud Latency Prediction

We first evaluate accuracy of WebPerf's predicted cloud latency under six what-if scenarios. For each scenario and for each application, WebPerf predicts a distribution of the cloud latency of loading the index page. To quantify the accuracy of WebPerf's prediction, we compare predicted latency distribution with ground truth latency distribution measured by actually deploying the applications under target what-if configurations. Given predicted and ground truth latency CDFs F_1 and F_2, we compute the *relative error distribution*—the distribution of vertical deviations ($|F_1(x) - F_2(x)|$) of two CDFs, and report statistics such as maximum, mean, and median of the deviations. (Note that the maximum of the relative error distribution is the D statistic used for Kolmogorov–Smirnov test for comparing two distributions.) Ideally, the relative error statistics should be close to zero.

We present detailed results only for SocialForum; results for other applications are summarized in Figure 12. Unless indicated otherwise, SocialForum is deployed in an Azure US West datacenter and clients load its specified page in a Chrome browser from California.

Scenario 1: *What-if the Redis cache is upgraded from the original Standard $C0$ tier to Standard $C2$ tier?* Figure 11(a) shows distributions of original latency (with $C0$ Redis), ground truth latency (with $C2$ Redis), and predicted latency (for $C2$ Redis). The result shows that upgrading Redis from $C0$ to $C2$ significantly changes cloud latency, and hence simply using the baseline performance of $C0$ tier as

[4]A pseudonym for the actual product.

Application	Description	Azure services used
SocialForum	A production-ready Microsoft application that provides Instagram-like social functionalities and allows users to create new accounts, create/join forums, post/share/tag/search pictures and comments, *etc.*	Blob storage, Redis cache, Service bus, Search, Table
SmartStore.Net [39]	An open source e-commerce solution that includes all essential features to easily create a complete online shopping website. It offers a rich set of features to handle products, customers, orders, payments, inventory, discounts, coupons, newsletter, blogs, news, boards and much more.	SQL
ContosoAds [12]	A classified advertisement website, similar to `craigslist.org`	Blob storage, Queue, SQL, Search
EmailSubscriber [15]	An email subscription service, similar to `mailchimp.com`, that allows users to subscribe, unsubscribe, and send mass emails to mailing lists	Blob storage, Queue, Table
ContactManager [11]	An online contact management web application, similar to `zoho.com/contactmanager`, that allows users organize to contacts	Blob storage, SQL
CourseManager [13]	A course management website, similar to `coursera.org`, that allows instructor course creation, student admission and homework assignments.	Blob storage, SQL

Table 3— *Third party applications used in our case studies and the Azure services they talk to.*

a prediction for the new tier will be inaccurate. As shown, WebPerf's prediction is very accurate: median, average, and maximum relative errors are 0.8% , 2.7% , and 18.3% respectively. We also used WebPerf to predict performance for two additional scenarios: upgrading Redis tier from $C0$ to $C6$ and upgrading the front-end web server tier from $A1$ to $A3$. The median relative errors of predictions for these two scenarios are 0.8% and 1.7% respectively.

Scenario 2: *What-if the front-end of SocialForum is replicated to two locations: US East and Asia East?* The backend still remains at US West. We configured WebPerf's profile dictionary with latency models for SocialForum's backend APIs when frontend and backend are deployed in the same datacenter (*e.g.,* US West). The models were then added with RTT distribution between backend and new frontend location (*e.g.,* US East). This helped us avoid profiling latency models for all possible combinations of frontend and backend locations. WebPerf's end-to-end prediction is quite accurate for both the locations, with median, mean, and maximum relative errors $< 4\%$, $< 2\%$, and $< 15\%$ respectively.

Scenario 3: *What-if SocialForum's reads/writes data of size X from/to blob storage?* We used X = 14KB, 134KB, 6.8MB, 12MB and 20MB. We configured WebPerf with offline profiles of blob storage API latencies for contents of different sizes. Figure 11(c) shows the CDFs of predicted and ground truth latency distributions for $X = 6.8MB$. The median errors for all values of X are below 9%.

Scenario 4: *What-if other collocated applications interfere with SocialForum?* Azure does not guarantee performance isolation for free tiers. We deployed SocialForum in a free tier and let other collocated applications create CPU pressure, using CPU loads of $10\%, 20\%, \ldots, 80\%$. The median relative errors for all the scenarios were $< 9\%$.

Scenario 5: *What-if N users concurrently load SocialForum webpage?* For this scenario, we configured WebPerf with profiles for API latencies for n concurrent API calls, for different values of n. WebPerf uses the dependency graph of the web request to determine n_c, the maximum number of concurrent execution of a I/O call c during each web request. Then, WebPerf uses c's latency profile under $N \times n_c$ concurrent calls. We used WebPerf to predict latency under N = 10, 20, 30, 40, 50, and 60 concurrent requests. Figure 11(d) shows the CDFs of distributions of ground truth and predicted cloud latencies for 30 concurrent requests. For all values of N, median prediction errors were $< 10\%$. end-to-end median prediction errors were $< 8\%$.

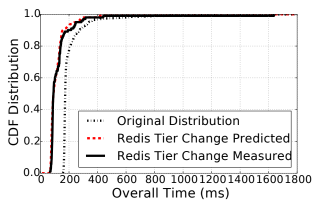
Scenario 6: *What-if a replicated frontend fails?* For this scenario, we replicated SocialForum's front end on two web servers and placed them behind a load balancer. We then used WebPerf to predict cloud latencies if one of the web servers dies. WebPerf assumes that when one web server fails, all user requests are routed to the live web server and hence its load effectively doubles. Thus, WebPerf predicts response times under a $2\times$ concurrent user requests (similar to the last scenario). We conducted the experiments with 10, 30, and 60 concurrent user requests. In all the cases, median relative prediction error was $< 9\%$.

Other applications: We conducted the above predictions for all the applications in Table 3. For consistency, we used the same set of scenarios across apps. In Scenario 1, we upgraded their frontends from the lowest tier to a mid tier. In Scenario 2, we replicated the frontend to Asia East. In Scenario 3, front-end retrieves 6MB data from backend blob storage. In Scenario 4, background CPU load is 70%. Scenario 5 uses 30 concurrent connections. Scenario 6 uses 60 concurrent connections and one of the two frontend fails.

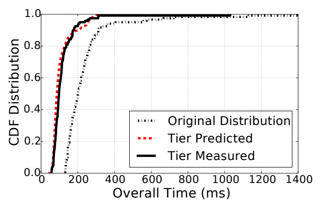
Figure 12(a) shows the median relative prediction errors for all applications and scenarios. Overall, errors are small ($< 7\%$), indicating that WebPerf is able to predict cloud latencies of a wide range of applications under the what-if scenarios we considered.

6.3 End-to-end Latency Prediction

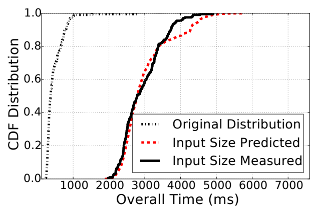
We now evaluate how well WebPerf predicts end-to-end latency (§ 5.1). We used all applications and scenarios used for cloud latency prediction above. To quantify prediction error, we obtained ground truth end-to-end latencies by accessing the index page of the applications in a Chrome browser from California. The cloud application was hosted in an Azure US West datacenter, and we used the network RTT distribution between the client and server as the network latency model. The relative error of the predictions (Figure 12(b)) for all applications and scenarios is, overall, small (median error $< 7\%$). Thus, WebPerf can also predict



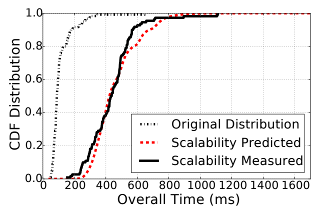
(a) Redis tier changed



(b) Web Server Tier Upgrade



(c) Input size changed



(d) Scalability

Figure 11—*Distributions of predicted and ground truth latencies for SocialForum under various what-if scenarios.*

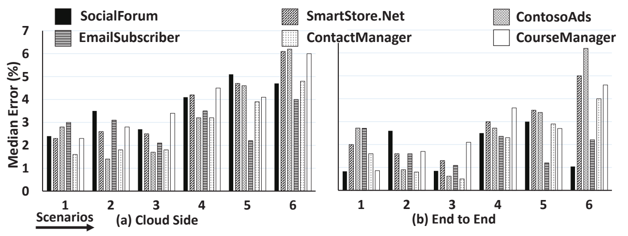


Figure 12—*Median prediction errors for 5 apps and 6 scenarios.*

end-to-end latency, given a good model for network latency between client and server and a WebProphet-like tool that can predict client-side latency.

Interestingly, WebPerf's end-to-end prediction is slightly more accurate than cloud-side prediction. This is because all the what-if scenarios we considered affect cloud-side latencies only, and offline cloud-side latency profiles have higher uncertainty than baseline latencies. In contrast, client-side prediction relies on baseline HTTP request latencies only, and hence is more accurate.

6.4 Comparative Analysis

Across all applications and scenarios, WebPerf is accurate when dependency graphs are small, and latency profiles are less variable and mostly application-independent.

Dependency graph complexity. The more I/O nodes in a dependency graph, the more the potential prediction errors. ContactManager has simple dependencies, and hence prediction for it is more accurate.

Profile variability. Profile variabilities manifest as prediction errors. In all the apps we evaluated, SocialForum's baseline profiles had the least variability (since only this app was deployed in dedicated VMs with performance isolation). This helped WebPerf generate a better prediction for SocialForum, despite its highly complicated dependency graph (each request had 100+ I/O calls). Over all the scenarios, Scenario 4 and 6 had higher prediction errors due to high variabilities in latency profiles used for these scenarios. In Scenario 6, frontend latency was highly variable under a large number of concurrent connections.

Application independence of profiles. Almost all APIs in our applications were application-independent. The only exceptions were ContosoAds and CourseManager, which use expensive SQL queries on medium sized tables. Other applications use small SQL tables, so latency does not change much with tier change (Figure 2). Latencies of queries to medium/large SQL table change as tiers change, and hence using baseline latencies as estimates resulted in large errors. The problem can be avoided by using workload hints, as shown next.

6.5 Using Workload Hints

We now show a few scenarios where workload hints from developer help WebPerf to improve its prediction.

Queries to large SQL tables. CourseManager issues a Join query on medium-sized SQL tables (10K rows). Using baseline latencies, WebPerf's prediction had a large error (median error 28.9%), because baseline latencies are not good estimates across tiers for large tables (Figure 2). However, with workload hints on specific query and table size, WebPerf could reactively build profiles for the given query and table size and lower error to 4.6%.

Nondeterministic caching behavior. Latency distributions of some APIs can be highly bimodal due to item popularity. For example, latency of accessing an item in Azure CDN depends on whether it is already in the cache, which in turn depends on application's workload. WebPerf's profiling recognizes such bimodal latency and maintains multiple (two in the CDN example) latency distributions as the profile for the API. Using a workload hint on the cache miss rate, WebPerf can combine these different distributions (using Monte Carlo simulation [26]). We profiled Azure CDN's data access API and evaluated a scenario with 1% cache miss rate. Median prediction error was 3.2%. Without the miss rate hints, the error was $> 40\%$.

Sharding Table storage. For SocialForum, we considered "what-if the Table is sharded into two". By default (without workload hints), WebPerf's prediction algorithm assumes that sharding will result in half the load on each shard, and predicts total latency based on Table storage's latency under half the load. We considered two workloads: (1) uniform workload where requests access both shards with roughly equal probability, in which case WebPerf's prediction was very accurate (median error 1.7%) (2) skewed workload where all requests accessed one shard, and since it received the full load. Without workload hints, WebPerf's estimation (assuming half load on each shard) has a median error of 90.1%; however, with the hint that the workload is highly skewed towards only one shard, prediction became fairly accurate (median error 5.9%).

6.6 Additional Results

Effect of concurrency limits. For ContosoAds, we considered "what-if we downgrade the SQL server from Standard to Basic tier, with 50 concurrent requests". The Basic tier only supports a maximum of 30 concurrent connections, and additional connections are queued. WebPerf's prediction is

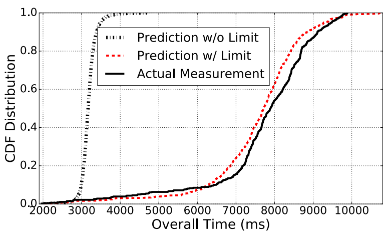


Figure 13—*Prediction for ContosoAds with concurrency limit.*

aware of such limits (§ 4.3) and in our experiment, it could make accurate prediction, as shown in Figure 13 (median error 2.1%); in contrast, ignoring such limit would have produced a median error of 87.3%.

Sources of prediction errors. At a high level, WebPerf has two sources of errors: API latency models and the prediction algorithm. To isolate errors introduced by the prediction algorithms alone, we start with execution trace of a request and use the prediction algorithm with true latencies of compute and I/O calls in the trace. We then compute the relative error of the predicted latency and the true total latency of execution trace. We repeat this for all requests used in our case studies. We find the average error to be 0.4% across all requests, with 0.3% median and 1.1% maximum error. The error is very small compared to the errors of the statistical models (mean and median errors ranging from 0.4%-6.5% and 0.5%-4.4% respectively), as shown in Figure 1. This suggests that WebPerf's prediction latency might be further improved by using more sophisticated models or more data on which models are built.

Impacts of measurement optimization. Finally, we evaluate our optimization algorithms described in §5.2. We use 3 distinct requests for the SocialForum website, containing 9 distinct API calls to 5 different cloud services. The average time for three requests are 1.3s, 3.8s, and 0.5s respectively. We use a time budget of 2 minutes for all measurements.

We compare our algorithm with two baseline schemes: 1) Round Robin (RR): all requests are repeated in round robin fashion for 2 minutes, 2) Equal Time (ET): all requests are allocated an equal time of 40 seconds each. For RR, each request was executed 22 times. For ET, requests 1, 2, and 3 are executed 30, 10, and 80 times. On the other hand, with our optimization algorithm, requests 1, 2, and 3 are executed 31, 19, and 15 times respectively. It collected more samples for APIs with high latency variability. For each API, we compared all predicted latency distributions with the ground truth latency distributions. The (mean, median, maximum) relative errors of our optimized algorithm for all APIs were $(1.8\%, 1.68\%, 10.1\%)$, while for RR and ET, the errors are $(5.55\%, 2.8\%, 21.0\%)$ and $(6.68\%, 2.4\%, 18.4\%)$ respectively. To achieve a similar accuracy as WebPerf, RR and ET need 2.6 min and 3.4 min respectively, more than 30% over the designated time. The results demonstrate a significant benefit of our optimization algorithm.

WebPerf overhead. To quantify the overhead, we fix the web applications' web server tier. In our experiment, our web server tier is standard tier. The average instrumentation time for all the six applications is 3.1s. WebPerf's instrumentation runtime overhead is lightweight - on average, it increases the run time by 3.3%. In the prediction stage, the overhead of obtaining the profile data is negligible as the profiling data are stored in Azure table. We expect some moderate overhead for the prediction algorithm's operation on the distribution; the average prediction time for all six applications' what-if scenarios is around 5.6s. We believe these overheads are reasonable: within tens of seconds, WebPerf is able to predict the performance for web applications under various scenarios quite accurately.

7. RELATED WORK

Performance prediction. Ernest [44] can accurately predict the performance of a given analytics job in the cloud. In contrast, WebPerf focuses on web applications that, unlike analytical jobs, are I/O intensive and are increasingly written using the task asynchronous paradigm. WISE [42] predicts how changes to CDN deployment and configuration affect service response times. Unlike WebPerf, WISE targets CDNs. Mystery Machine [10] uses extensive cloud-side logs to extract the dependency and identify the critical paths. It targets cloud providers, and hence uses extensive platform instrumentation. In contrast, WebPerf targets third-party developers, and relies on instrumenting app binaries alone. Both WISE and Mystery Machine use purely data-driven techniques In contrast, WebPerf has less data when an application is being deployed, and hence it uses a combination of instrumentation and modeling for prediction.

WebProphet [25] can predict webpage load time, but it focuses only on end-to-end prediction. In contrast, WebPerf considers both cloud-side prediction and end-to-end prediction. WebProphet extracts (approximate) dependencies between web objects through client-side measurements, while WebPerf extracts accurate dependencies by instrumentation. CloudProphet [24] is a trace-and-replay tool to predict a legacy application's performance if migrated to a cloud infrastructure. CloudProphet traces the workload of the application when running locally, and replays the same workload in the cloud for prediction. Unlike WebPerf, it does not consider what-if scenarios involving changes in configurations and runtime conditions of web applications. Herodotou et al. [20] propose a system to profile and predict performance of MapReduce programs; however, the technique is tightly integrated to MapReduce and cannot be trivially generalized to a web application setting.

Dependency analysis. Closest to our work is AppInsight [32, 33], which can automatically instrument .NET application binaries to track causality of asynchronous events. However, it does not support the *Async-Await* programming paradigm. To our knowledge, no prior instrumentation framework can accurately track the dependency graphs for applications written in this paradigm.

Significant prior work has focused on dependency graph extraction [36, 16, 8, 38, 35, 5, 27, 47]. Systems that use black-box techniques [36, 47] fail to characterize the accurate dependencies between I/O, compute, and other components, and hence can result in poor prediction accuracy. Systems that modify the framework [16, 8, 38, 35] are hard to deploy in cloud platforms. And techniques that require developer effort [5, 27] are hard to scale. There are also systems that track dependencies of network components [46, 9]. WebPerf differs from these frameworks in its target system (cloud-hosted web application).

Webpage Analysis. [17, 45, 14, 19, 30] measure website performance by analyzing the request waterfall. Pagespeed [17] analyzes the webpage source and proposes content optimization. Unlike WebPerf, these systems only focus on the client-side view and not the server-side.

8. CONCLUSION

We presented WebPerf, a system that systematically explores what-if scenarios in web applications to help developers choose the right configurations and tiers for their target performance needs. WebPerf automatically instruments the web application to capture causal dependencies of I/O and compute components in the request pipeline. Together with online- and offline-profiled models of various components, it uses the dependency graphs to estimate a distribution of cloud and end-to-end latency. We have implemented WebPerf for Microsoft Azure. Our evaluation with six real web application shows that WebPerf can be highly accurate even for websites with complex dependencies.

Acknowledgements. We thank the reviewers and our shepherd Ion Stoica for their comments that helped improve the technical content and presentation of the paper.

9. REFERENCES

[1] ANTS Performance Profiler. http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/.

[2] Asynchronous Programming with Async and Await. https://msdn.microsoft.com/en-us/library/hh191443.aspx.

[3] Amazon Web Services Asynchronous APIs for .NET. http://docs.aws.amazon.com/AWSSdkDocsNET/V2/DeveloperGuide/sdk-net-async-api.html.

[4] Azure Web Extension Gallery . https://www.siteextensions.net/packages.

[5] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online Modelling and Performance-aware Systems. In *HotOS*, 2003.

[6] R. Bellman. Dynamic programming and Lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 1956.

[7] Callback Hell. http://callbackhell.com/.

[8] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *IEEE DSN*, 2002.

[9] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *USENIX OSDI*, 2008.

[10] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale Internet services. In *USENIX OSDI*, 2014.

[11] Contact Manager. https://azure.microsoft.com/en-us/documentation/articles/web-sites-dotnet-deploy-aspnet-mvc-app-membership-oauth-sql-database/.

[12] Contoso Ads. https://azure.microsoft.com/en-us/documentation/articles/cloud-services-dotnet-get-started/.

[13] Course Manager. https://www.asp.net/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application.

[14] Dareboost Website Analysis and Optimization . https://www.dareboost.com/en/home/.

[15] EmailSubscriber. https://code.msdn.microsoft.com/Windows-Azure-Multi-Tier-eadceb36/sourcecode?fileId=127708&pathId=382208951.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *USENIX NSDI*, 2007.

[17] Google PageSpeed Insights. https://developers.google.com/speed/pagespeed/insights/.

[18] C. M. Grinstead and J. L. Snell. *Introduction to probability*. American Mathematical Soc., 2012.

[19] GTmetrix Website Speed Analysis . https://gtmetrix.com/.

[20] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB*, 2011.

[21] High Scalability: building bigger, faster, more reliable websites. http://highscalability.com/.

[22] JetBrains dotTrace. http://www.jetbrains.com/profiler/.

[23] Y. Jiang, L. Ravindranath, S. Nath, and R. Govindan. Evaluating "What-If" Scenarios for Cloud-hosted Web Applications. Technical Report MSR-TR-2016-36, Microsoft Research, June 2016.

[24] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang. CloudProphet: Towards Application Performance Prediction in Cloud. In *ACM SIGCOMM*, 2011.

[25] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang. WebProphet: Automating Performance Prediction for Web Services. In *USENIX NSDI*, 2010.

[26] C. Z. Mooney. *Monte carlo simulation*. Sage Publications, 1997.

[27] New Relic. http://newrelic.com/.

[28] Node.js. https://nodejs.org/en/.

[29] NUGET Package. https://www.nuget.org/packages.

[30] Pingdom Website Speed Test. http://tools.pingdom.com/fpt/.

[31] F. Pukelsheim. *Optimal design of experiments*. SIAM, 1993.

[32] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *USENIX OSDI*, 2012.

[33] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *ACM SOSP*, 2013.

[34] Redis Cache. https://azure.microsoft.com/en-us/services/cache/.

[35] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *USENIX NSDI*, 2006.

[36] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *ACM WWW*, 2006.

[37] D. J. Rumsey. *Statistics For Dummies, 2nd Edition*. Wiley, 2011.

[38] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google research*, 2010.

[39] SmartStore.Net. http://www.smartstore.com/net/en/.

[40] Stackify. http://stackify.com/.

[41] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Practical Aspects of Declarative Languages*. Springer, 2011.

[42] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar. Answering what-if deployment and configuration questions with wise. In *ACM SIGCOMM CCR*, 2008.

[43] TechStacks. http://techstacks.io/stacks.

[44] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *USENIX NSDI*, 2016.

[45] Webpage Test. http://www.webpagetest.org/.

[46] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *USENIX NSDI*, 2011.

[47] Z. Zhang, J. Zhan, Y. Li, L. Wang, D. Meng, and B. Sang. Precise request tracing and performance debugging for multi-tier services of black boxes. In *IEEE DSN*, 2009.