Rain: A Workload Generation Toolkit for Cloud Computing Applications

Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, David Patterson
{abeitch@, bran_liu@, yung@, rean@eecs, fox@eecs, pattrsn@eecs}.berkeley.edu
RAD Lab, EECS Department, UC Berkeley

Abstract

Existing workload generation tools are not flexible enough to generate workloads that vary in three key dimensions relevant for making resource allocation decisions for Cloud Computing applications – variations in the *amount* of load, variations in the *mix* of operations performed by clients (e.g. changes in reads vs. writes or customer usage-patterns) and variations in the popularity of the data accessed, i.e., *data hotspots*.

In this paper we present Rain, a workload generation toolkit designed to address this lack of flexibility. Rain allows for the use of probability distributions as first-class elements that describe variations in the three key dimensions of workload Its architecture supports multiple workload generation strategies (open-loop, closed loop and partly-open loop), easy extensibility via user-defined request generators targeting new systems/applications – we demonstrate targeting the Olio Web 2.0 application. Rain also supports trace generation, which allows it to be used in conjunction with high-performance load-replay systems, e.g., httperf.

1 Introduction

The advent of cloud computing has enabled easy access to large-scale cluster computing at low costs. This ease of adoption attracts a diverse set of users with varying application profiles, resource demands and usage patterns. A plethora of services, ranging from Web 2.0 portals to big-data scientific or financial computations as well as data-centric decision engines rely on cloud computing infrastructure for scaling and system management.

A consequent challenge for application developers and service providers is the need to anticipate changes in workload patterns and adapt quickly to or at least account for underlying system bottlenecks and behavior. For example, a current event or new feature added to a website can cause changes in usage patterns, which result in a workload spike that affects data hot spots and cluster utilization. It is therefore important for application developers and cloud providers to evaluate a system's ability to cope with variability and equip themselves to make effective resource allocation decisions in the presence of of such variability.

There are three major types of variability we have observed in modern applications running on cloud infrastructure:

- **Cost-benefit analysis based load variations**: The primary constraints of the cloud computing user are to a) use as few resources as necessary to meet demand and b) use resources as efficiently as possible. An advantage of using pay-as-you-go execution environments is that they scale transparently to the end user and seamlessly adapt to varying amounts of load. The application developer may decide to scale up or scale down by adding or removing machines to adjust for changes in the *amount* of load and reduce resource wastage. As the cloud infrastructure provider, the decision to provision more machines may be to prevent service level agreement (SLA) violations. On the other hand, scaling down may improve cluster utilization, reduce power consumption and minimize cost.

- **Externally imposed behavioral variations**: A side effect of the ease of deployment on cloud computing environments is that it enables a high churn rate in the software stack. For example, the overhead of adding features to a website are low, and thus more common. Consequently, there is high variability in user behavior as users react to new features and popular trends. Such behavioral variations increase variability in the *mix* of operations performed by the user, resulting in continually-revised resources allocations to anticipate and react quickly to the evolving workload. For example, transitioning from a workload dominated by write operations

to one dominated by read operations may result in a larger subset of resources being dedicated to servicing reads via more data replicas. Such resource provisioning decisions are in addition to and independent of the fluctuations in the *amount* of workload. Furthermore, different cloud applications solicit different usage patterns, which pose more challenges to the cloud provider to make efficient tradeoffs across multiple services hosted on their cluster.

- **Data popularity and hot spots**: A consequence of highly variable usage patterns is that data must be stored in a manner that enables efficient access based on application requirements. For example, applications with sensitive data require stringent privacy guarantees, real-time applications would benefit from replication to minimize data access times, and applications that allow multiple simultaneous writes (such as shared calendars) require stringent consistency guarantees. Such application-specific guarantees must be satisfied despite usage and system load variations. Simultaneously, applications must have the option of relaxing their requirements to reduce infrastructure cost.

A key aspect in responding to these variations in load is the quality of the system's reaction. This task requires mechanisms that quickly detect different workload variations, decide whether the variation is transient or sustained and respond to either minimize the negative effects of transients or adequately adapt to the new workload regime. While the above three factors affect one another, the mechanisms to handle them can be independently implemented. User behavioral variation can be limited by setting thresholds on the number of concurrent users allowed and what fraction of them are exposed to which features. Data hot spots can be alleviated by increasing the number of redundant copies, and accounting for data access locality.

In this paper, we propose a flexible, representative and adaptable workload generation framework. We describe specific design challenges and requirements posed by cloud computing applications. We identify and address the architectural limitations of existing workload generation tools that currently preclude their usage in more sophisticated workload generation scenarios and we argue that statistical modeling and probability distributions are an invaluable tool for driving workload generation tools that can accommodate "what-if" scenario-based decision making.

Our work makes three contributions:

1. First, we present the architecture of Rain and highlight how its design departs from that of existing workload generation tools and the flexibility this departure offers us in enacting sophisticated workload scenarios.

2. Second, we describe the implementation of Rain and describe the creation of a workload generator for a Web 2.0 application, Olio, which is used as the target system in the Cloudstone Web 2.0 benchmark [13].

3. Third, we provide an open source toolkit for researchers and developers to use and extend and discuss how Rain can be used to target other Cloud Computing applications or environments.

The remainder of this paper is organized as follows. §2 presents related work. §3 provides an overview of Rain, its architecture and the motivations behind our design choices. §4 discusses the creation of a workload request generator for a typical cloud computing application – the Olio a Web 2.0 application. §5 evaluates the workload generation capabilities of Rain as it generates load against the example target system. Finally §6 summarizes our results and contributions and outlines future work.

2 Related Work

Traditionally, workload generators are an after-thought for systems researchers and application developers alike, and the need for such a tool has been satiated with regression tests, back of the envelope calculations, and feature-specific stress tests. However, such methods do not scale to the churn rate, load variations, system behavioral patterns and changes in usage behavior that typify applications deployed in the cloud. Contemporary workload generation tools are focused on evaluating peak-workload performance, however, cloud computing applications, must exhibit good peak-performance *and* be able to adapt to variations in workloads. Current workload generation tools fall short of allowing application developers to explore how these workload variations affect their systems and how they can improve their system's reactions. Furthermore, most existing workload generation frameworks do not distinguish between workload request generation and request execution, making it difficult to separate request generation from load delivery as we explain further in Section 3.

There are several simple application-specific workload generators that allow users to generate workload that conforms to a predetermined distribution. For example, SURGE generates URL references based on file size, request size, and file popularity distributions [10], SLAMD allows users to generate workload for Java-based networked applications, specifically LDAP directory servers [5], StreamGen allows users to gener-

ate distributed events and data streams [12], and Harpoon mimics internet traffic and generates representative background traffic for application and protocol testing [14]. All the above examples require the user to specify file/usage parameter distributions, which is infeasible if the expected workload pattern does not fit into one of a set of pre-defined distributions.

Although exceptions rather than the norm for testing software, Optixia [3] and Hammer [1] are hardware-based traffic generators that provide a dedicated hardware device for creating and transmitting network traffic patterns, including internet traffic, VoIP, and telephone call generation. While such hardware-based generators are excellent for load testing through maximizing request volume, they tend to be very expensive to acquire, and are not easily adaptable to new request types and usage patterns.

Several workload generators have been coupled with applications to form benchmarks, predominantly for web services. SPECweb generates http requests to static content based on request distributions, but such simplistic workload does not befit Web 2.0 applicaions [6]. TPC-W is an attempt to represent more complex web service requests, and comprises an online bookstore that allows web serving, browsing and shopping cart facilities [8]. However, the content is static and the requests are generated based on a predetermined mix matrix of request transition probabilities. Furthermore, the TPC-W workload generator requires a significant amount of state to be maintained per simulated user, and thus its scalability is limited by the backend database. RUBiS [4] provides an eBay-like auction portal, which introduces more request complexity than TPC-W, but suffers from the same limitations as TPC-W with respect to mix matrix-based request generation and poor scalability. Cloudstone [13] is an attempt to adapt these kinds of benchmarks to Web 2.0 applications. Cloudstone consists of Olio a social event calendar web application and Faban, an open source workload generator. Faban can emulate thousands of concurrent Olio users interacting with the application. Like TPC-W and RUBiS, user behavior is described by a markov mix matrix, however, whereas Faban supports variable load during a benchmark run, the user behavior remains fixed.

While all the above workload generation tools are widely used in their specific application domains, they do not perform well in the presence of a black-box system. They require a certain degree of prior knowledge of the system, for which the workload generator is custom-configured.

3 Rain: Olio Workload Generator

Rain is a statistics-based workload generation toolkit that provides a thin, reusable, configuration and load-scheduling wrapper around application-specific workload request-generators, which can easily use parameterized or empirical probability distributions to mimic different classes of load variations.

The load scheduling harness takes care of managing variations in the amount of load and the mix of operations, while application-specific request generators are responsible for adapting to changes in the mix of operations and simulating data hotspots (if required).

Our architecture differs from that of existing workload generation tools in two significant ways.

First, we decouple request generation from request execution. To our knowledge, this a departure from existing workload generation tools, e.g., the load generators used in TPC-W [8], Rubis [4], Cloudstone (Faban) [13], which couple request generation and request execution together. Separating request generation and request execution allows us to produce traces that may be consumed by third-party high-performance load-delivery clients such as httperf [2]. The separation affords us the flexibility of evolving the request generators independently of the load delivery harness and it opens up the possibility of using more computationally intensive request generators (e.g., where generating the next request involves a series of simulation steps) without skewing the runtime measurements of the target system since requests can be generated, saved and (re)played at a later time.

Second, we remove the *thread-affinity* between requests and their execution by parameterizing requests with all the state they need for execution when they are generated. In existing workload generation tools, the thread that causes a new request to be generated is also the thread that executes it. This unnecessarily restricts the workload generation tool to a closed-loop workload strategy (a thread creates a new request, executes it and waits on its completion before creating the next request). In Rain, the requests (Operations) produced by a Generator are implemented according to the Command pattern [11]. This design choice allows an Operation to be executed by any thread, provided it can be suitably parameterized by the Generator that creates it. As a result, the thread that generates an Operation does not necessarily have to execute it, which enables us to cover the full spectrum of workload generation strategies (closed-loop, open-loop or partly-open loop).

3.1 Architecture

Figure 1 shows the architecture of Rain. There are five major components:

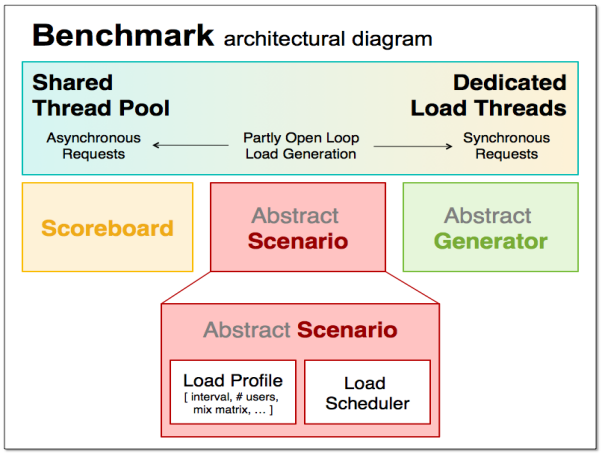

Figure 1: Rain Architecture Diagram

- A *Benchmark* orchestrates an entire workload experiment. It loads the Scenario, initializes the Scoreboard where results are collected and summarized, the shared threadpool and the dedicated load threads that run the request *Generators* – Generators produce the Operations to be executed. At the end of the experiment, the Benchmark terminates all the components and presents the results collected on the Scoreboard.

- A concrete *Scenario* contains all the configuration parameters for an experiment, including but not limited to, the maximum number of users to emulate, the duration of an experiment, the ramp up and ramp down interval and the sequence of *LoadProfiles* to enact during a run.

  At a bare minimum, a LoadProfile contains an interval (in seconds), the number of users active and the name of a mix matrix describing the behavior of each user. LoadProfiles are consumed by the dedicated load threads and there are MaxUsers dedicated load threads spawned at the start of an experiment and assigned ids $(1, ..., MaxUsers)$.

  The LoadProfile $[100, 500, default]$ would be interpreted as: for the next 100 seconds, run 500 active users (dedicated load threads with ids < 500 remain active, while all others sleep) and each user follows the behavior codified in the mix matrix labeled "default". During an experiment sequences of LoadProfiles allow us to vary the number of users and their behavior over time.

- A concrete *Generator* creates requests for a single active user. It is initialized with a Scenario and determines the next request (a concrete Operation) based on data about the previous request (if necessary) and any other internal criteria.

- The *Scoreboard* is a thread-safe, singleton object, where all Operations write (drop off) their execution results. These results are processed and summarized by a background worker thread and presented at the end of an experiment. The Scoreboard also arranges for Operation traces to be written out to persistent storage.

- The *Shared threadpool* and *dedicated load threads* are available for executing the Operations produced by request Generators (if required). A Generator is owned by a single dedicated load thread, however, the Operations it produces may be executed by some other thread (e.g. a thread in the shared threadpool).

Model of Operation Concrete Generators – one for each emulated user – are initialized using the Scenario. A dedicated thread is associated with each generator. This dedicated thread requests the *next operation* from the generator and either waits synchronously on the operation to execute or hands the operation off to the shared threadpool for execution and immediately requests a new operation from the generator, to simulate an asynchronous request.

Operations, once they finish executing, write their execution results (success/failure, performance metrics, etc.) on the scoreboard and are then are discarded. Figure 2 shows the relationship and APIs of the major actors in the Rain toolkit.

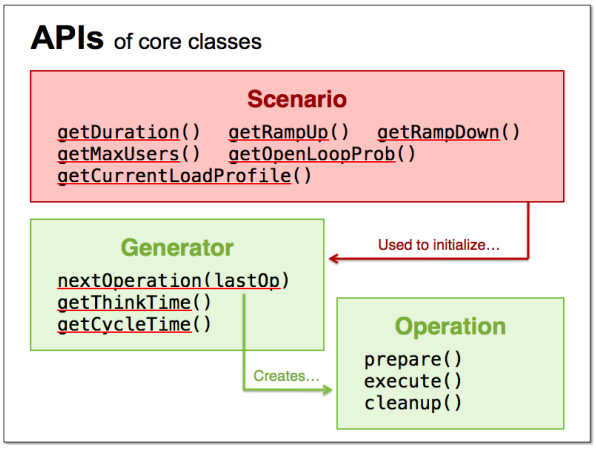

Figure 2: Rain Core APIs

HomePage	0.00	0.11	0.52	0.36	0.00	0.01	0.00
Login	0.00	0.00	0.60	0.20	0.00	0.00	0.20
Tag Search	0.21	0.06	0.41	0.31	0.00	0.01	0.00
Event Detail	0.72	0.21	0.00	0.00	0.06	0.01	0.00
Person Detail	0.52	0.06	0.00	0.31	0.11	0.00	0.00
Add Person	0.00	0.00	0.00	0.00	1.00	0.00	0.00
Add Event	0.00	0.00	0.00	1.00	0.00	0.00	0.00

Figure 3: Default Olio Mix Matrix

4 Case Study

4.1 Creating the Olio Workload Request Generator

Olio [7] is the target application of the Cloudstone Web 2.0 benchmark developed jointly by Sun and members of the RAD Lab at UC Berkeley. The Olio application implements a social-event calendar web application (in PHP and Ruby) that provides functionality representative of Web 2.0 applications – user-generated metadata, social networking functions and a rich AJAX-based GUI [13].

Olio users can perform a number of workflows (operations) including: adding events, viewing event details, attending events, adding new users, viewing user profile details, searching for user-generated metadata (tag search), logging into of the application, and browsing the homepage.

The Olio distribution includes a UIDriver that is used by the Faban workload generator to generate and execute requests against an Olio deployment. The UIDriver uses a markov mix matrix to govern the behavior of simulated Olio users and the series of operations they perform.

To create the Olio workload request generator for Rain, we refactored the monolithic UIDriver into a hierarchy of seven concrete parameterizable Olio operations (HomePage, Login, TagSearch, EventDetail, PersonDetail, AddEvent, AddPerson), which extend the base Operation class in Rain (Figure 4). We also create an OlioGenerator class that creates instances of these operations, based on the transition probabilities specified in markov mix matrix (Figure 3) and knowledge of the previous type of operation.

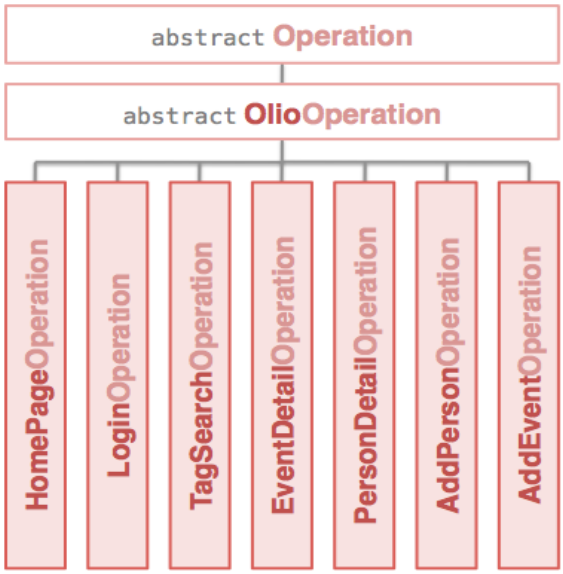

Figure 4: Olio Workload Class Diagram for Rain

5 Evaluation

5.1 Olio

Our Rain prototype is implemented in 1813 lines of Java code. To evaluate our prototype we ported the workload driver (UIDriver) for the Apache Olio application, found in the Faban workload generator [13].

We create an Olio request-generator (*OlioGenerator*) by porting Faban's UIDriver, and then drive load against a small cluster running Olio and deployed on EC2. Our goal is to identify the performance bottleneck of the cluster using Rain and our OlioGenerator. Creating the OlioGenerator and porting the seven Olio operations required 2841 lines of Java code.

Since we are interested in identifying the performance limit/bottleneck of the cluster we eliminate the think times between individual user requests. This choice allows us to use a smaller set of concurrent users since the absence of think times increases the effective load delivered.

Our experimental setup uses 4 EC2 instances. 1 m1.small running Nginx and HAProxy, 1 c1.xlarge running Rails, 1 c1.xlarge running MySQL v5.0.28. We emulate 1 to 150 concurrent users performing Olio operations as fast as they can (no think time between operations).

Figures 5 and 6 show the cluster performance in Olio operations per second and HTTP requests per second for 1 to 150 concurrent users. The peak performance of our

simple cluster is $\sim 34 \pm 3.06$ Olio operations per second, which is equivalent to $\sim 740 \pm 65.86$ HTTP requests per second (we conducted 5 repetitions and include the 95% confidence intervals for our measurements).

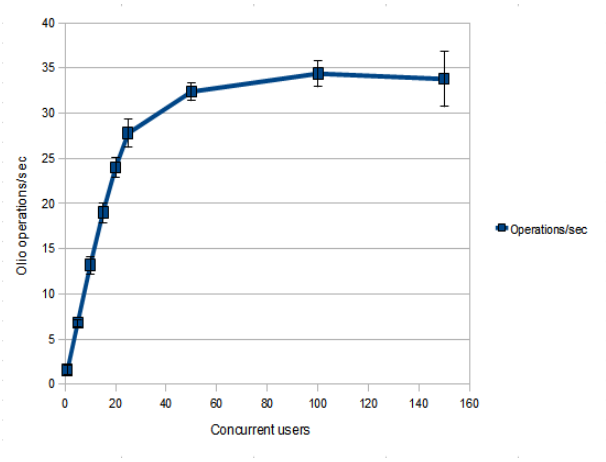

Figure 5: Rain fixed load results (Olio Operations/sec)

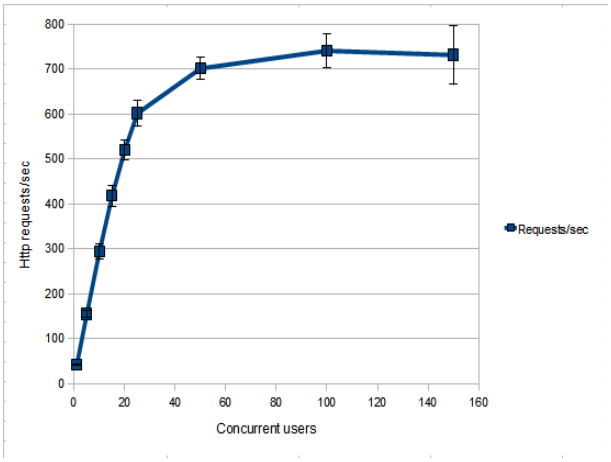

Figure 6: Rain fixed load results (HTTP requests/sec)

Figures 7 and 8 show the results an experiment where the number of users is varied every 60 seconds. These results are reconstructed from the traces from five runs, collected at 10 second-intervals. As a result of the differences between the load increment-interval and the data collection interval, we omit the error bars in this figure.

6 Conclusion

In this paper we highlight three key variations in cloud computing application workloads (amount, mix and data hotspots). We identify limitations of existing workload

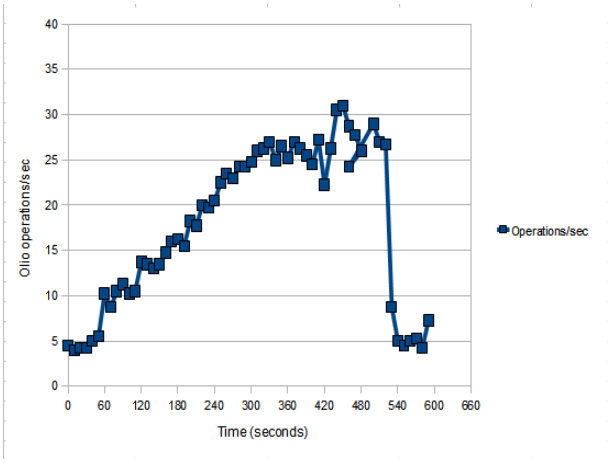

Figure 7: Rain variable load results scaled from 1 to 150 concurrent users and then back down to 1 user (Olio Operations/sec)

generation tools that artificially restrict the choice of workload generation strategy and the workload variations that can be reproduced – tight coupling between request generation and request execution and an implicit close-loop load generation restriction. We present architectural solutions to those limitations and implement a toolkit Rain, designed specifically to address these limitations.

We implement an example workload request generator for a typical cloud computing application (the Olio Web 2.0 application) and show that our prototype workload generator (i.e., Rain + Olio request generator) can capture the three aspects of load variations we describe.

For future work we are constructing additional workload request generators for two other cloud computing applications – MapReduce batch processing applications and SCADS (a Scalable Consistency-Adjustable Data Storage system for interactive applications) [9]. We are also working towards a release of the Rain toolkit, including example workload request generators, under a GPL license for researchers and application developers to use.

Finally, we have added support in Rain for generating workloads for multiple tenants/applications in single a experiment using the abstraction of multiple parallel tracks of load profiles. This feature raises methodological questions about generating representative multi-tenant/application workloads that we are interested in exploring further.

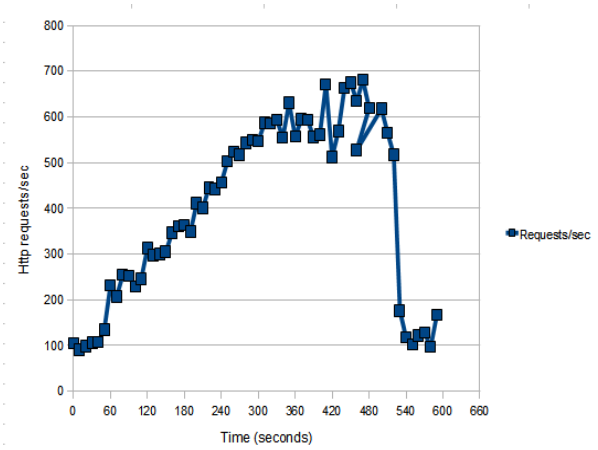

Figure 8: Rain variable load results scaled from 1 to 150 concurrent users and then back down to 1 user (HTTP requests/sec)

7 Acknowledgments

This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240. Special thanks to our undergraduate assistants - Aaron Beitch, Timothy Yung, Brandon Liu.

References

[1] Hammer. www.empirix.com.

[2] httperf Homepage. http://www.hpl.hp.com/research/linux/httperf/.

[3] Optixia. www.ixiacom.com/products/optixia.

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

[5] Slamd. http://www.slamd.com.

[6] SPECweb. http://www.spec.org/web2005.

[7] The Apache Olio Project. http://incubator.apache.org/olio/.

[8] TPC-W. www.tpc.org/tpcw/default.asp.

[9] *SCADS: Scale-independent storage for social computing applications* (01/2009 2009).

[10] BARFORD, P., AND CROVELLA, M. E. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of Performance '98/SIGMETRICS '98* (July 1998), pp. 151–160. Software for Surge is available from Mark Crovella's home page.

[11] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software (Hardcover)*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[12] MANSOUR, M., WOLF, M., AND SCHWAN, K. Streamgen: A workload generation tool for distributed information flow applications. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 55–62.

[13] SOBEL, W., SUBRAMANYAM, S., SUCHARITAKUL, A., NGUYEN, J., WONG, H., PATIL, S., FOX, A., AND PATTERSON, D. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *CCA '08: Proceedings of cloud computing and its applications* (2008).

[14] SOMMERS, J., KIM, H., AND BARFORD, P. Harpoon: a flow-level traffic generator for router and network tests. *SIGMETRICS Perform. Eval. Rev. 32*, 1 (2004), 392–392.