Demystifying Cloud Benchmarking

Tapti Palit
Department of Computer Science
Stony Brook University
tpalit@cs.stonybrook.edu

Yongming Shen
Department of Computer Science
Stony Brook University
yoshen@cs.stonybrook.edu

Michael Ferdman
Department of Computer Science
Stony Brook University
mferdman@cs.stonybrook.edu

***Abstract*—The popularity of online services has grown exponentially, spurring great interest in improving server hardware and software. However, conducting research on servers has traditionally been challenging due to the complexity of setting up representative server configurations and measuring their performance. Recent work has eased the effort of benchmarking servers by making benchmarking software and benchmarking instructions readily available to the research community.**

Unfortunately, the existing benchmarks are a black box; their users are expected to trust the design decisions made in the construction of these benchmarks with little justification and few cited sources. In this work, we have attempted to overcome this problem by building new server benchmarks for three popular network-intensive workloads: video streaming, web serving, and object caching. This paper documents the benchmark construction process, describes the software, and provides the resources we used to justify the design decisions that make our benchmarks representative for system-level studies.

I. Introduction

The past two decades have seen a proliferation of online services. The Internet has transitioned from being merely a useful tool to becoming a dominant part of life and culture. To support this phenomenal growth, the handful of computers that were once used to service the entire online community have transformed into clouds comprising hundreds of thousands of servers in stadium-sized data centers. Servers operate around the clock, handling millions of requests and providing access to petabytes of data to users across the globe.

Driven by the constantly rising demand for more servers and larger data centers, academia and industry have directed significant efforts toward improving the state-of-the-art in server hardware architecture and software design. However, these efforts often face a number of obstacles that arise due to the difficulty in measuring the performance of server systems.

While traditional benchmarks used to measure computer performance are widely accepted and easy to use, benchmarking of server systems calls for considerably more expertise. Server software is complex to configure and operate, and requires *tuning* the server's operating system and server software to achieve peak performance. Whereas typical datasets for traditional benchmarks are intuitive to identify, the datasets for server systems are diverse and include not only the contents of the data, but also the frequency and the access patterns to that data. Moreover, while the performance of traditional benchmarks is easily defined as the time taken to complete a unit of work, quantifying the performance of a server is inherently more challenging because it must take into account the *quality of service* (latency of requests) and not just the peak raw throughput.

Several recent projects [1], [2] have begun to address the challenges of benchmarking servers faced by the research community. These efforts have made great strides in identifying the relevant server workloads and codifying performance measurement methodologies. They collected the necessary benchmarking tools and have been disseminating instructions for setting up server systems under test, configuring software that simulates client requests, and generating synthetic data sets and statistical distributions of request patterns. As a result, the effort needed to benchmark server systems for a typical researcher has gone down drastically.

Unfortunately, although acknowledging their benefits, we identified a core drawback of the existing server benchmarking tools. While they provide the software and installation directions, the existing benchmark suites do not readily provide justification for the myriad decisions made in the construction of the benchmarks. The existing benchmark tools are essentially a black box; users of these tools are called upon to implicitly trust the decisions that were made in their design. The design choices for the existing benchmarks and the justifications for these choices are not clearly cited and therefore do not allow users to make a decision regarding the actual relevance of these benchmarks. Moreover, when some of the design choices become dated, a revision of the benchmarks to make them representative of the new server environment becomes necessary. However, in the existing benchmarks, these decisions remain undiscovered without deep investigation of the tools.

This paper chronicles our experience in setting up new server benchmarks with explicitly justified design choices. In this work, we concentrate on benchmarking three of the most popular network-intensive online applications:

- *Video Streaming* services dominating the Internet network traffic [3].
- *Web Serving* of dynamic Web2.0 pages performed by the most popular websites in the world [4].
- *Object Caching* of data used extensively in all popular cloud services [5].

To the extent practical, we document the tools that we use, leveraging prior work and expertise. Wherever possible, we explicitly describe the decisions that were made in the construction of our benchmarks and cite the motivation and the sources

that led to those decisions. Finally, we have integrated our benchmarks with the CloudSuite [1] and led the re-engineering of all benchmarks in CloudSuite 3 using Dockers [6], a mechanism that makes it easy to use the benchmarks and documents all of the configuration choices in clear machine-readable form.

The rest of this paper is organized as follows. We motivate replacing the existing benchmarks in Section II and describe the key considerations in server benchmark design in Section III. In Section IV, we describe our benchmarks and detail the design decisions we made in their creation. Section V present the overview of our Docker-based benchmark deployment setup. We present several surprising results and compare our benchmarks to prior work in Section VI. We review the related work in Section VII and conclude in Section VIII.

II. Motivating the Change

We begin the presentation of our workloads by motivating our decision to create new benchmarks. After using the existing CloudSuite benchmarks [1] in several studies, we realized that a number of decisions made in these benchmarks do not match the current state of practice. These realizations initially motivated us to fix some of the problems we observed. However, after fixing a number of problems with the Web Serving and Data Caching benchmarks, we realized that it is easier to use the knowledge we gained in this process (and some of the existing tools) to re-implement the benchmarks, forcing us through the process of considering many of the implicit design decisions along the way.

It is worth noting that the focus of our new benchmarks is system-level evaluation and performance measurement. Micro-architectural study of the CloudSuite workloads demonstrated that vastly different cloud workloads have similar micro-architectural characteristics [1]. In this work, we target making the system-level behavior representative of real-world behavior, while the micro-architectural behavior is expected to remain largely similar to the original CloudSuite workloads. In Section VI-D we confirms that, from a micro-architecture perspective, the workloads are practically the same.

A. Video Streaming

In the past, various streaming application layer protocols were used for delivering audio and video content over IP networks. Protocols such as the Real Time Streaming Protocol (RTSP) were designed for media streaming and specifically targeted the needs of this application. However, RTSP is a stateful protocol that keeps track of user sessions on the server, making it difficult to scale out the service horizontally. Moreover, the common RTSP implementations used UDP, which presented problems for clients behind network address translation (NAT) proxies [7]. Because of this, video-streaming websites such as YouTube and Netflix built their streaming services on the Hypertext Transfer Protocol (HTTP) [8]. Using HTTP allows stateless servers that can be easily scaled out in modern environments and permits the use of existing battle-tested high-performance web servers such as NGINX [9] as the core of these streaming services. Because the existing video-streaming workload from the CloudSuite [1] was built for RTSP, both the server and the client infrastructure of the benchmark needed to be replaced to make the workload representative of modern video-streaming services.

B. Web Serving

In older generation websites, the user's interaction consisted primarily of reading content and periodically clicking on URLs that changed the reading content of the entire page. Although submission of web forms was supported, this was a relatively rare operation, and the submitted data rarely if at all affected the website content, both for the user submitting the form and for other users. On the other hand, Web2.0 websites are more responsive and provide a richer user experience, similar to GUI-style applications. These websites cause the browser to dynamically fetch small chunks of data and integrate it into parts of the webpage, instead of loading the entire webpage content at once. Because of this, the amount of data transferred per request is significantly less than older web applications where the whole page had to be reloaded for each request.

The CloudSuite [1] Web Serving workload was intended to represent the new Web2.0 paradigm. However, the Olio project [10] used for this benchmark was written as a benchmark and was never in use in a production environment. In fact, a large fraction of the requests made in this benchmark were requests for static images, a task that is commonly outsourced to content distribution networks (CDNs) in production systems. Moreover, the fraction of dynamic requests in the benchmark corresponded to the situation circa 2008. We replaced the benchmark website with a more representative package of a production system, with a larger fraction of small dynamic requests, that aligns more closely with modern practices. Importantly, while we deemed the server-side software inappropriate, the Faban [11] framework used to emulate web clients in the original CloudSuite benchmark is a good fit and we leveraged it in the creation of our own web serving benchmark.

C. Object Caching

The Memcached server software used for the Data Caching benchmark in the CloudSuite [1] remains one of the most popular and most-widely used cloud applications [5]. However, as we started heavily using this benchmark, we discovered a number of drawbacks in the software that simulated client requests to the server. As we delved deeper into understanding the benchmark behavior, we realized that its behavior was not representative of real Memcached deployments. For example, while UDP-based GET requests are common in real deployments [12], the client included in the benchmark suite threw an "Unimplemented" exception when configured for UDP. We were interested in using the benchmark because it included the "Twitter" dataset, which was appealing because it appeared to represent a real-world use case of the application. However, upon further investigation, we discovered that the client included a bug that performed incorrect scaling of the dataset. We therefore decided to pursue implementing our own

client that included support for UDP requests and correct dataset scaling.

III. Key Considerations for Benchmark Design

Benchmarking cloud applications has unique challenges over traditional benchmarking. Traditional benchmarking measures wall-clock time, which is the time needed to complete an operation, with no other considerations. On the other hand, because the ultimate goal of cloud applications is end-user satisfaction, measuring the performance of server applications requires also taking into account the Quality-of-Service (QoS) as a proxy for end-user satisfaction. QoS is typically specified as a maximum latency L for a percentile P of all requests, meaning that $P\%$ of all requests must be completed within latency L. As a result, measuring performance under QoS constraints implies iteratively applying different loads to find the peak load under which the QoS requirements are met. Importantly, such performance measurements imply that the measured system is under-utilized at its peak performance (higher utilization would yield higher throughput, but the QoS requirements would not be met).

Moreover, benchmarking cloud applications requires faithfully mimicking the behavior of real clients. In case of video-streaming servers, some videos are accessed more frequently than others; Web2.0 client requests are dominated by small dynamic AJAX requests; object cache systems handle a wide range of key and object sizes and requests, all of which can affect the observed server throughput [13]. To properly measure a server's performance, the statistical distribution of request sizes and popularity emulated by the benchmarking tool must be representative of real-world setups [12], [14]. When designing benchmarks for cloud applications, one must ensure that the request mix closely resembles real clients.

Finally, we note that benchmarking is typically performed in a lab environment, where client machines connect to the server over a high speed network. Each client machine simulates the behavior of hundreds or thousands of real clients. Therefore, care must be taken to ensure that each of the simulated clients is operating in an environment similar to that of real clients. We found this of particular concern for video streaming, where network traffic is bursty because of different media-streaming clients requesting chunks of the file at different points in time. Without artificially-imposed limits, it is possible for a single simulated client to gain access to the entire bandwidth of the high-speed link. Such behavior is not representative of real client environments that experience a range of limited and varying bandwidth capabilities. The benchmarking environment must provide a realistic setup for simulated clients to ensure that the behavior that the server exhibits resembles the behavior observed in real deployments.

IV. Our benchmarks

We base our benchmarks on several production applications: the NGINX-based HTTP video-streaming server, the Elgg social networking engine [15], and the Memcached key-value store.

For benchmarking video-streaming applications, we deploy a video-streaming server on NGINX with video files in different video resolutions. We ensure that the distribution of video files and video resolutions is similar to that of popular video-streaming services such as YouTube. We use the methodology described in [16] for our work, basing our client on httperf [17], a tool from HP Labs. Accesses to videos follow a popularity distribution [18]; we therefore take into account the difference in the popularity of the videos when simulating the clients.

Elgg is a production-ready, actively used and developed social networking engine, which has similar functionality to Facebook. The bulk of the workload that dynamically generates web responses is performed by PHP, one of the most-popular platforms for developing dynamic websites [4]. Inspired by the CloudSuite [1], we use the Faban framework to develop a benchmark for Elgg. Faban uses a light-weight Java thread for each client, allowing the simulation of thousands of clients on a single machine with relatively low memory requirements.

Memcached is a key-value store server used by some of the most popular websites [12] and is a representative object caching application. We developed a new Memcached benchmarking client called *Memloader*. Memloader allows users to specify arbitrary key-size, value-size and item-popularity distributions using a dataset file and provides first-class support for UDP requests. With these features, Memloader can accurately emulate real-world clients. In addition to Memloader, we developed a set of automation scripts to simplify the task of benchmarking a Memcached server.

A. Video-streaming benchmark

Streaming video is a content delivery method in which videos are continuously delivered to the client, instead of being downloaded at once. The video is progressively transferred to the user as it is being watched.

Two techniques – pacing and chunking – are used to stream video over HTTP. The video-streaming client's requests to the server are "paced," ensuring that the video-streaming client retrieves only the part of the video that will be played back in the near future. HTTP/1.1 Range Requests are used to request one "chunk" of the video at a time. At the start of playback, the streaming video client prefetches a few chunks of video content. It then waits for these chunks to be viewed by the user. As the fetched chunks are consumed and the buffers are emptied, the next chunk in the sequence is fetched.

Millions of videos are uploaded to video-streaming services, but only a small fraction of them are popular. The popularity of videos follows a Zipf distribution [18]. A small percentage of the videos are very popular and are accessed regularly, while the majority of the videos are accessed rarely. The request patterns of our benchmark reflect this popularity distribution.

The bitrate at which content is encoded determines the stream quality. Encoding content at a higher bitrate yields a higher quality stream, increasing the amount of data that needs to be downloaded to play the video. If a low-bandwidth client attempts to view a high-quality video, the speed at which the buffers are replenished could be lower than the speed at which

the video is viewed. The client will experience pauses during viewing due to "buffering," causing the viewer to become frustrated and potentially stop the video playback.

To solve this problem, the quality of the video stream is varied depending on the available network bandwidth of the client. Videos uploaded to the video-streaming service are stored encoded at different bitrates, creating different quality versions of the same video, each of a different size.

1) System to benchmark – NGINX based Video streaming: Although our infrastructure can benchmark any HTTP-based video-streaming server, we select the NGINX server. NGINX is an event-driven HTTP server built around an asynchronous I/O architecture. Event-based I/O systems allow a single application thread to handle multiple file descriptors, unlike the thread-based I/O models (employed by servers such as Apache) which require one thread or process per client. A single NGINX thread can service many client connections, making NGINX highly scalable and performant. For these reasons, prominent video-streaming services use NGINX for content distribution [19].

We enable the following NGINX configuration options:

1) *Sendfile*: Sendfile is a system call that directly copies contents from one file descriptor to another within the kernel. Enabling sendfile speeds up copying of data between the video file descriptor and the network socket descriptor by avoiding unnecessary memory copies and context switches.
2) *Epoll event mechanism:* Under Linux, NGINX supports the select, poll, and epoll event mechanisms. We use the Epoll mechanism because it has the best performance and scalability [20].
3) *Persistent connections:* HTTP persistent connections enable one TCP connection to be reused for multiple requests. This mitigates the overhead of initiating and tearing down TCP connections during video playback. By default, in HTTP/1.1, all connections are persistent for a fixed duration specified by a timeout value. The default installation of NGINX specifies this value as five seconds. We set the timeout to 60 seconds, which exceeds the inter-chunk interval at all bitrates and allows the server to shut down connections only if they are no longer used by the client for video streaming.

2) Benchmarking Tools: We base our benchmarking work on the methodologies and tools presented in [16]. The benchmarking tools consist of an enhanced httperf client that acts as the benchmark driver, a file-set generator, and a httperf log generator. The file-set generator generates the videos on the video-streaming server. The httperf log generator generates a log simulating the sequence of URLs accessed by the clients. The httperf client executes the benchmark by replaying the log of requests against the server under test.

a) Benchmarking Driver: Our benchmarking client driver is based on httperf [17], which was enhanced in [16]. The httperf tool, originally developed at HP Labs, measures web server performance by simulating the behavior of many concurrent web users. For our benchmark, we developed a workload generator *videosesslog*, based on the design of wsesslog.

Wsesslog allows the specification of the behavior of individual client sessions through its workload generator. A session comprises of a sequence of bursts, spaced out by the user think-time. Each burst is one or more requests to the server. The wsesslog workload generator allows the specification of parameters of these client sessions, such as the sequence of URIs to access and the think-time between requests. In our workload, the think-time parameter is used to simulate "pacing" by separating requests for consecutive chunks of a video, simulating gradual viewing.

We specify a time-out period for each request. If the response to the request is not received within the timeout period, the request is considered as causing buffering. Buffering corresponds to frustrated users of the video-streaming service – in other words, these are violations of Quality-of-Service.

In addition to the features supported by the wsesslog workload generator, the videosesslog workload generator supports multiple input log files. The user can specify multiple input logs and a probability distribution. Videosesslog generates requests from each of these input logs, according to the specified probability distribution, enabling the user to specify the percentage of requests that will be generated from each of the input logs. The user can specify requests for different video qualities in different input logs and then specify the probability distribution, described in Table I, for the ratio of accesses from each of these input logs. Videosesslog allows binding each input log to a different local IP on the client machine. Doing this enables dummynet [21] rules that limit network bandwidth at each local IP to simulate realistic network conditions.

The httperf tool gathers statistics, such as the percentage of connections which timed-out, average reply rate, and average request rate, and summarizes and displays these statistics at the end of the benchmark execution.

b) Request mix generation tool: As discussed in Section IV-A, video popularity follows a Zipf distribution. A small subset of videos are more popular and are accessed more frequently. The make-zipf [17] program is used to generate the list of videos and corresponding videosesslog logs, such that the requests reflect the Zipf distribution, with a configurable Zipf exponent. Similar to [16], we use Zipf exponent -0.8 for our experiments. This list contains the name of the video files, their duration, and popularity rank. The list of videos is then read by the gen-fileset tool, which creates the video files.

c) File-set generation tool: The file-set generator, gen-fileset [17], reads the list of videos and creates the files on the video-streaming server. The number of files generated, and therefore the dataset size, are configurable. Video-streaming servers store different quality versions of the same video. We support Low Definition (240p), two Standard Definition (360p, 480p), and High Definition (720p) resolutions. We generate all videos at Low Defintion (240p) and two Standard Definition (360p, 480p) resolutions. Additionally, High Definition (720p) resolution is generated for 20% of the videos. The size of a

TABLE I
BROADBAND CLIENTS - BANDWIDTH DISTRIBUTION

Bandwidth	Percentage of users
Above 15 Mbps	19%
10 Mbps - 15 Mbps	20%
4 Mbps - 10 Mbps	34%
1 Mbps - 4 Mbps	27%

video file depends on its bitrate and its duration. We use the bitrates suggested by YouTube [22] for our calculations.

3) Benchmark Setup: We simulate clients with different bandwidth capabilities. From Akamai [23], we obtain the distribution of the network speed for worldwide broadband users in 2015. The percentage of clients with different bandwidth capabilities are presented in Table I.

We use dummynet [21] to emulate clients with different bandwidth. Dummynet is a network emulation tool that can perform network bandwidth shaping. It can filter packets based on any combination of parameters that identify a TCP connection (Source/Destination MAC-address/IP-address/Port-number). These filters can be used to forward the packets through a virtual pipe, which is configured with attributes such as a bandwidth cap or traversal latency. We configure multiple IP aliases on each of the client machines and use dummynet to filter packets on the basis of these IP aliases into different pipes. On each pipe, we configure a bandwidth limit for each TCP connection that passes through the pipe.

B. Web2.0 benchmark

Web2.0 is a set of principles that guided the shift in the direction of web development after the year 2001 [24]. Web2.0 websites have certain characteristics that cause these workloads to be different from the workloads of older generation websites. Older generation websites typically served static content, while Web2.0 websites serve dynamic content. Web2.0 websites have richer user-interfaces that engage the user more frequently than older websites.

A Web2.0 website delivers a service or platform, unlike traditional websites. For example, the social networking site Facebook delivers a social networking platform to the users. Most of the content on these websites consists of data provided by the users of the website and not by the web developer. The content is dynamically generated from the actions of other users and from external sources, such as news feeds from other websites. Because of this, writes to the backend database are frequent and the data written is consumed by other users.

1) System to benchmark – Elgg: The Elgg social networking engine is a Web2.0 application developed in PHP, similar in functionality to the popular social networking engine Facebook. Elgg is currently used by the Australian Government, the New Zealand Ministry of Education, Wiley Publishing, the University of Florida, and many other organizations [25].

Elgg allows users to build a network of associations or friends. It provides a platform for the users to share content. Elgg includes a microblogging platform, called Elgg Wire, which can be used to share text, image, or video content with other users. Posting content on this microblogging platform makes it available to be read by other users. This is similar to Facebook's popular Wall functionality. Every user has a live feed of content shared by their network of friends. This live feed is called Elgg River. Several plugins exist to custom-tailor the base functionality with additional features desired for a particular installation.

The Elgg platform and the available plugins allow the user to carry out a variety of operations, such as sending and receiving chat messages, posting on Elgg Wire, and retrieving the latest posts. These operations are AJAX based, sending and receiving many small requests. The workload is dominated by these frequent AJAX requests.

Elgg uses PHP as its server-side scripting language and uses MySQL as its database backend. Similar to the setup used at Facebook [12], we enable Memcached to cache the results of database queries. We enable the Zend Opcache, which is a PHP "accelerator" commonly used in production environments. We change the default storage engine of MySQL to InnoDB to support a large number of concurrent reads and writes, needed to support many concurrent users to the website.

2) Benchmarking Tools: We use the Faban [11] framework to develop our benchmark for Elgg. Faban is a Java-based benchmark development and execution tool with two main components: Faban Driver framework and Faban Harness.

Faban Driver is a framework that provides an API that can be used to quickly develop a benchmark. A benchmark driver is defined by the operations it runs. The request mix for the benchmark is specified by the list of operations to be performed and the probability of each operation.

The Faban Harness comprises an Apache Tomcat instance that hosts a web application which automates deploying and running the benchmark. At the end of the benchmark run, a report is generated that contains statistics such as the success/failure count of each operation, the response time, and the number of quality-of-service violations.

3) Faban Benchmark Details: Our benchmark takes into account the fact that different operations occur with different frequencies. In the Faban driver for our benchmark, we specify a function for each of the operations in Table II. In the mix, we assign higher probability for more common operations, such as updating the live feed, posting on walls, and sending and receiving chat messages. We assign a lower probability for operations such as login and logout, reloading the home page, and creation of new users, as these are carried out less frequently. Also, each operation is assigned an individual QoS latency limit. Table II shows our request mix and the QoS latency limit for each operation. We derive these values by extrapolating Facebook's page load time, which is reported as 2.93 seconds by Alexa [26].

We specify a Quality-of-Service (QoS) requirement of 95% for our benchmark. 95% of all operations performed must meet the QoS limit specified for that operation. If less than 95% of the operations meet the QoS latency limit, the Faban driver deems the benchmark run as failed.

TABLE II
ELGG – REQUEST MIX, QUALITY-OF-SERVICE LIMITS

Request	Percentage	QoS (in seconds)
Create new user	0.5%	3
Login existing user	2.5%	3
Logout logged in user	2.5%	3
Access home page	5%	1
Wall post	20%	1
Send friend request	10%	1
Send chat message	17%	1
Receive chat message	17%	1
Update live feed	25.5%	1

Our benchmark clears the transactional data between each run to avoid any possible performance degradation due to large database tables and to ensure that the execution environment is similar from one run to the next.

4) User prepopulation tool: Before running the benchmark, we must prepopulate the database with Elgg users. These are simulated clients who will log in to the system and perform operations. We developed the UserSetup utility for this purpose. This utility can create a configurable number of users and forward their login credentials to the Faban benchmark driver. When the benchmark is launched, each benchmark client thread logs in with one of these users' credentials and proceeds to perform the operations described in Table II as that user. The number of pre-generated users determines the maximum number of client threads that can be launched, in turn determining the maximum scale of the benchmark.

C. Object Caching benchmark

In order to improve performance, web servers use object caching systems to cache the results of expensive computations, thus making object caching an important workload to study. In this section, we present a benchmark for Memcached, which is a popular, open-source, object caching system.

1) System to benchmark – Memcached: Memcached is a popular object caching system, which is typically used by web servers to cache the results of expensive database queries. It is a completely in-memory key-value store. It supports both TCP and UDP protocols. Memcached typically acts as an object-cache for web servers, and a single web request can result in many Memcached requests. Therefore, to avoid delaying web requests, Memcached requests need to be serviced with low latency. The latency requirements for Memcached are typically 1 to 2ms. A single Memcached request requires very little processing. So a Memcached server can serve over a million requests per second, which is a significantly higher throughput than those of other workloads. The benchmark is capable of benchmarking a single Memcached server or a cluster of Memcached servers. The dataset can be either replicated or sharded across multiple servers.

Our Memcached benchmarking tool comprises two parts. The first is Memloader, an efficient C++ program for traffic generation and performance statistics collection. The second is the benchmarking harness, which is a collection of scripts for automating the benchmarking task.

2) Benchmarking Tools – Memloader: Memloader emulates a large number of virtual clients that perform requests to a Memcached server. Each virtual client independently generates requests and examines responses. If a cluster of servers or server processes are benchmarked, a virtual client can send requests to multiple servers. By default, Memloader spawns one worker thread per CPU core. Half of the worker threads are dedicated to sending requests. The other half are dedicated to receiving responses. The separation reduces interference between request and response activities, enabling precise timing of request generation and accurate statistics of the response latencies. A request-sending thread can send requests on behalf of multiple virtual clients. Similarly, a response-receiving thread can receive responses on behalf of multiple virtual clients. Memloader threads are pinned to CPU cores to avoid overhead from unnecessary thread migrations. Memloader can send requests using ether TCP or UDP.

Key-size, value-size and item-popularity distributions can be specified by providing Memloader a dataset file, where each line represents a data item and specifies that item's key size, value size, and popularity. By populating a dataset file with appropriate records, any key-size, value-size and item-popularity distributions can be achieved. Memloader can synthesize a large dataset from a small dataset file. Conceptually, this is done by replicating the same dataset multiple times for use by the virtual clients. The actual implementation stores the small file in memory and performs the replication on the fly, ensuring a small memory footprint. All virtual clients use the same dataset.

The performance measured by Memloader is based on the specified target latency (e.g., 1ms). Memloader reports the percentage of requests that completed within the target latency. Memloader also reports the average latency, the throughput, the number of outstanding requests, and the hit-ratio. If detailed analysis of a server's performance is needed, Memloader can output the complete response latency histogram.

3) Benchmarking Harness: To automate the benchmarking task, we provide a benchmarking harness, which is a set of scripts. Its main functions are system configuration, peak throughput seeking, and multi-client control.

The Memcached server under test is likely to use a high-performance NIC. In this case, the configuration of the server OS and NIC driver can have a significant impact on the measured performance. For example, to get the most out of RSS [27] support in the NIC, all CPU cores should participate in interrupt handling. If the NIC supports a TCP flow director [28], out-going packets of a TCP connection should be sent out through a single queue, so that the flow director can correctly associate the TCP connection with the queue. Also, the CPU core responsible for handling that TCP connection's incoming data should handle the associated queue's interrupts. If the purpose of the test is to measure a server's maximum performance, the frequencies of all server cores should be set to the maximum. Similarly, proper system configuration on the client side is necessary to avoid overloading the client machines, which would yield erroneous traffic generation patterns

and latency measurements. Managing these settings manually is a tedious and error prone task, especially when many machines are used in the benchmarking setup. To address this problem, the benchmarking harness automatically configures server and client machines with peak performance settings. We provide an example configuration harness for Intel Xeon E5v3 Linux servers with Intel 82599ES NICs.

When benchmarking a high performance Memcached server or a cluster of Memcached servers, a single client machine may be insufficient to drive the requisite load. To address this problem, the benchmarking harness supports deploying instances of Memloader across multiple client machines and coordinates their simultaneous execution. The harness parses outputs from all Memloader instances and aggregates the performance statistics.

A common goal for benchmarking a Memcached server is to find the server's peak throughput within QoS constraints. Manually re-running experiments with different throughputs to find the best performance that does not violate QoS requirements is a tedious task. The benchmarking harness can automatically perform a binary search for the peak throughput by repeatedly running the benchmark at different loads and automatically monitoring the QoS.

V. Docker deployment mechanism

Working with the CloudSuite 2 benchmarks, we found that the installation and configuration process is extremely complex and error-prone. The benchmarking software, system libraries, and Linux kernel often create complicated dependencies that must be maintained for the software to run correctly. However, as newer Linux distributions are released, support for older system libraries and kernels are gradually phased out, requiring the benchmark user to manually resolve dependencies. Moreover, not only did the process of following the installation instructions require significant time and effort, the provided instructions were outdated and often not applicable to the new Linux distributions.

To remedy this situation, we implemented our new benchmarks using Docker containers [6]. Docker containers parcel the benchmarks along with the complete filesystem needed to execute them. This includes all dependencies – the benchmarking software, system tools, and system libraries. Moreover, the Dockerfile not only serves as a quick and painless way to automatically recreate the benchmark setup, but it simultaneously serves as pedantic and precise documentation of the exact dependencies, installation procedures, and configuration settings of the benchmark.

The Docker containers for our benchmarks are released on Docker Hub [29], a free globally-accessible repository for Docker containers. The benchmark user can perform a "docker pull" to download these containers into their local machines. Once downloaded, the user can run the containers by issuing a "docker run" command. Dockers greatly simplify the benchmark deployment process, distilling it into two simple commands. Dockers enable benchmark installation and bring-up in seconds instead of days and ensure stable and consistent benchmark setups for each installation. Based on the positive experience we gained in bringing up our new workloads with Dockers, this approach to benchmark distribution has been adopted in CloudSuite 3 for all of the benchmarks.

VI. Results

In this section, we demonstrate several key aspects of our benchmarks (CloudSuite 3) by comparing them to their counterparts from CloudSuite 2. First, we examine the results of running the benchmarks on a typical server and compare three notable metrics: request mix, I/O utilization, and interrupt distribution. Next, we present two case studies that demonstrate how the key considerations for cloud benchmark design, outlined in section III, affect server characteristics and measured performance. Finally, using the Video-streaming and Memcached workloads, we show that, although the system-level behavior of these workloads is radically different, the micro-architectural behavior when running these benchmarks is similar to the previous generation CloudSuite workloads.

A. Comparison of Request Mix

We contrast the request mixes of our video-streaming and Web2.0 workloads with the CloudSuite 2 workloads for the respective applications to highlight the key system-level differences between them.

1) Request Mix for Video Streaming: As described in Section IV-A, different video-streaming clients have different bandwidth capabilities and the video stream quality is selected based on the bandwidth of each client. Moreover, only a fraction of all videos are available in a High Definition format. Because video resolution has a direct impact on the file size of the video, the request mix of the benchmarking tool must be similar to real-world situations in order to accurately represent the server and network characteristics.

Figure 1 compares the request mix of our NGINX-based benchmark with the request mix of the CloudSuite 2 RTSP-based benchmark. Our new benchmark supports four video qualities: 240p, 360p, 480p and 720p, whereas the older CloudSuite 2 benchmark supports three video qualities: Low (160p), Medium (240p), and High (360p). In the default configuration, the CloudSuite 2 benchmark accesses videos of all three qualities with equal probability. In contrast, our workload takes into account the varying bandwidth availability of broadband clients, as described in Table I, and mimics real-world behavior where not all videos are available in High Definition.

2) Request Mix for Web2.0 benchmarks: As discussed in section III, Web2.0 clients perform many small AJAX requests. Thus, for a large fraction of requests, the response comprises only a few bytes, which update a small part of the webpage, instead of kilobytes of data required to update the entire webpage. Because the number of bytes transferred per request determines the network characteristics, a workload to benchmark these applications must be representative of this reality.

We compare the request mix of a run of our Elgg benchmark with the request mix of CloudSuite 2 Olio benchmark. Figure 2 shows the comparison of the AJAX requests and requests that

Fig. 1. Request Mix – Video Quality

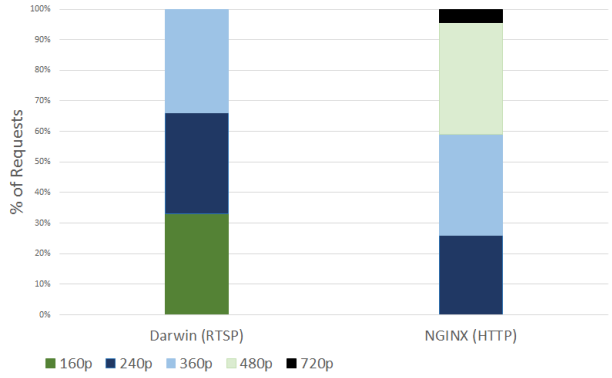
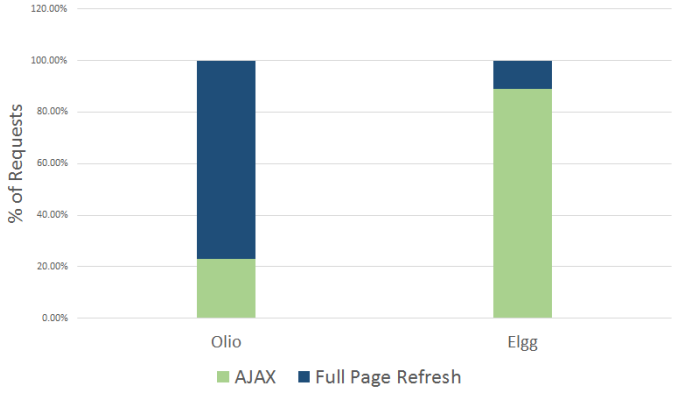


Fig. 2. Request Mix – AJAX vs Full Page Refresh



result in full page refresh. In Olio, 77% of all operations in the request mix result in full page refreshes. On the other hand, our Elgg benchmark consists of 89% of small, AJAX requests, which update a small part of the webpage, and only 11% of requests perform full page refresh.

B. *Video-Streaming – Effect of Request Mix on I/O Wait%*

In this section, we study the effect of the popularity distribution of a video-streaming request mix on the disk I/O wait time, to show how the request mix of a workload has an impact on server characteristics. As discussed in Section IV-A, a small percentage of videos on the video-streaming server are frequently accessed. This allows the operating system to cache these "hot" files, thus reducing the total number of disk accesses and speeding up reads of the less frequently accessed videos. Ultimately, this affects the Quality-of-Service of the video-streaming system.

For our experiments, we use a server and client machine with the configuration described in Table III. Our fileset consists of 8000 videos, each available in 240p, 360p, and 480p video resolutions. Additionally, 20% of the videos are available in 720p quality. The total size of the fileset is 1.1 TB.

For this experiment, we generate two request mixes – a popularity-aware request mix and a popularity-unaware request mix. The popularity-aware request mix takes into account the popularity distribution of videos, as described in IV-A, and the popularity-unaware request mix accesses all videos with an equal probability.

TABLE III
EXPERIMENTAL SETUP

CPU	Intel® Xeon® E5-2620 v3 @ 2.40GHz
Number of cores	12 (2 sockets, 6 per socket)
Hyperthreading	Off
RAM	64 GB
NIC	Intel 82599ES 10 Gbps
SAS Disk Array	10 disks, 15K RPM
OS	Ubuntu 14.04.3 (kernel 3.19.0-30)

Fig. 3. IO-Wait % for Popularity-Aware and Popularity-Unaware Mix

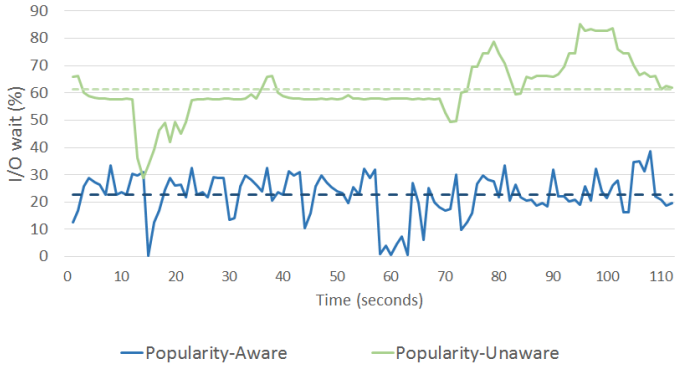


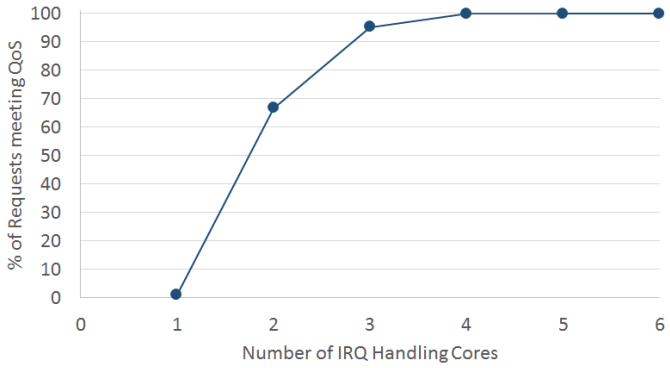
Figure 3 shows the I/O-Wait% for these configurations. The average I/O-Wait% for the popularity-aware request mix is 22.6%, while that of the popularity-unaware request mix is 61.2%. As a result of this, in case of the popularity-unaware request mix, 55% of the requests fail to meet the QoS requirements. In case of the popularity-aware request mix, only 4% of the requests fail to meet QoS. The system-level behavior is therefore greatly dependent on this parameter and should be selected to match real-world conditions.

C. *Memcached – Effect of Interrupt Distribution*

Prior work on Memcached servers has shown the importance of factors like value size, item popularity, using UDP, and others [13], [12]. In addition to having a realistic workload, it is important to configure the server under test in a way that ensures there are no artificial bottlenecks in the system. In this section, we demonstrate the effect of interrupt distribution on the server machine on the benchmark's result. A Memcached server handles hundreds of thousands of requests per second, which corresponds to a massive number of network interrupts on the server. To make sure a multi-core system is not bottlenecked on interrupt handling, the server NIC requires RSS support. However, having an appropriate NIC by itself is not sufficient. To properly configure a Memcached server with an RSS-capable NIC, interrupts must be evenly distributed among cores, which is often not the default behavior.

We demonstrate the impact of interrupt distribution using two machines with identical hardware and software configuration

Fig. 4. Memcached Performance and IRQ Handling



(Table III), connected by a 10G network. The server runs Memcached 1.4.25. The client sends 860K requests per second to the server through 600 TCP connections. The dataset comprises five million items with key size 40 bytes, value size 500 bytes, and uniform popularity. 80% of the requests are GET and 20% are SET. We use this simple workload to isolate and highlight the effect of interrupt distribution. The key size and value size are chosen according to [14]. On the server, only 6 cores (one socket) are used, the other 6 cores are halted.

Figure 4 shows the system performance, measured as a percentage of requests meeting the 1ms latency requirement, as we vary the number of server cores that handle receive interrupts. We find that the performance difference exceeds 33% as the number of interrupt-handling cores is varied from 2 to 6, demonstrating the magnitude of the impact that this parameter has on overall system throughput. Although the default configuration may perform some interrupt distribution to partialy mitigate this problem, our object caching benchmark demonstrates the configuration needed to equally distribute interrupts to achieve peak performance from the hardware.

D. Micro-architectural Features of Workloads

We demonstrated major differences in the system-level behavior of our benchmarks compared to CloudSuite 2. We now examine the micro-architectural behavior of two of the workloads, video-streaming and Memcached, with their CloudSuite 2 counterparts. Similar to [1], we concentrate on IPC, L1-Dcache and L1-Icache misses, and D-TLB and I-TLB misses.

For all experiments, we use one client and one server machine. The configuration for both the client and server are described in Table III. To ensure a fair comparison, we examine the server systems under identical CPU load. Figure 5 shows the comparison of the IPCs for the video-streaming and Memcached workloads. Prior work found that the IPC achieved by cloud workloads is relatively low [1]. We find that the IPC of our workloads and their CloudSuite 2 counterparts are practically the same, differing by less than 3%. Figures 6 and 7 further compare the micro-architectural behavior of the workloads with respect to the caches and TLBs. From these results, we conclude that the micro-architectural characteristics of the CloudSuite 2 workloads and our workloads are similar.

Fig. 5. IPC Comparison

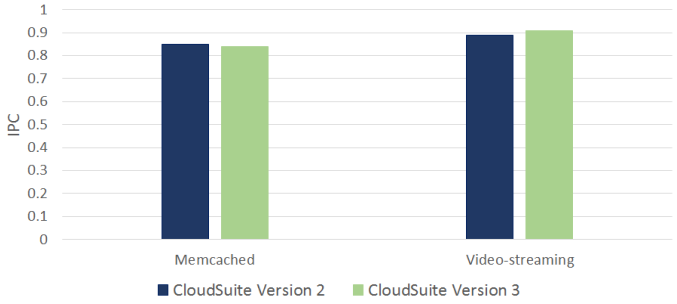
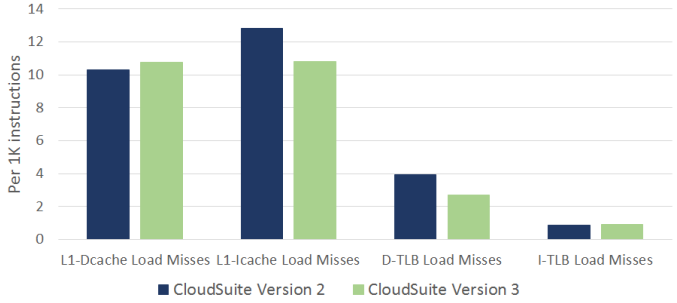
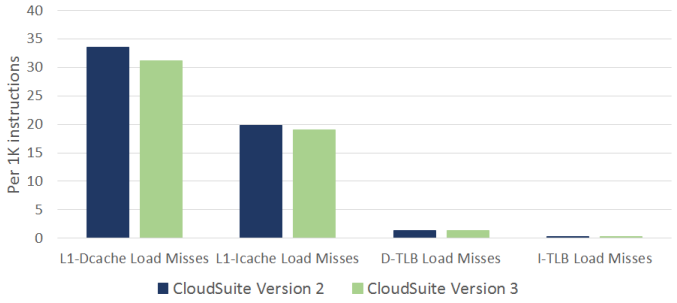


Fig. 6. Micro-architectural Features - Video-Streaming Workloads



Thus from the results described in sections VI-B, VI-C, and VI-D, we see that even though the micro-architectural characteristics of CloudSuite 2 and our new (CloudSuite 3) benchmarks are similar, the system-level behavior differs significantly. Thus, for the purpose of micro-architectual studies, both CloudSuite 2 and CloudSuite 3 workloads are equally suitable. However, for the purpose of system-level evaluation, CloudSuite 3 workloads are more suitable because they are more representative of real-life situations.

Fig. 7. Micro-architectural Features - Memcached Workloads



VII. Related Work

There has been considerable work related to benchmarking cloud workloads. We present a survey of the popular benchmarking tools for video streaming, Web2.0, and object caching (Memcached) applications.

A. Video streaming

The CloudSuite 2 includes a video-streaming benchmark based on the Darwin video-streaming server. Darwin serves

content using the Real Time Streaming Protocol (RTSP). However, today, popular video-streaming services, such as YouTube and Netflix, stream video using the HTTP protocol [8].

Benchlab [30] is a web application benchmarking framework that uses trace-replays on real web browsers and gathers statistics on both the client and the server. The Benchlab tool was extended into the Video-Benchlab suite [31] for benchmarking video-streaming servers. Unfortunately, because the tool launches a separate browser instance for each simulated user, and each browser instance uses a significant amount of memory capacity and CPU time, the number of clients that can be simulated by a single machine in a benchmarking setup is severely limited. The per-client resource requirements make Video-Benchlab impractical for studying servers under high throughput conditions.

Methodologies for generating HTTP streaming video workloads were presented by [16]. Our video-streaming benchmark makes extensive use of these methodologies and our video-streaming benchmarking tool is based on the source code provided as part of the work.

B. Web2.0

Olio was developed to benchmark a "typical" Web2.0 application, a social event calendar that allows multiple users to post social events, browse events, and "friend" other members. The CloudSuite 2 [1] benchmarking suite includes a Faban driver that measures the performance of Olio. However, the Olio application is retired, no longer supported, and remains an example of an outdated Web2.0 benchmark application that was never used in a production environment.

Benchlab [30] (mentioned above) was originally designed to benchmark websites, but suffers from the problem of high per-client resource requirements, which makes it unsuitable for launching thousands of simulated clients on a single machine.

SPECweb [32] is a popular SPEC benchmark for evaluating web server performance. The benchmark contains three workloads: Banking, E-commerce, and Support. These workloads are traditional web applications and don't have the characteristics of a modern Web2.0 website. Moreover, similarly to Olio, SPECweb is officially retired and is no longer maintained.

C. Object Caching

CloudSuite 2 contains a Data Caching benchmarking client [1] that can generate a workload based on a "Twitter" dataset. However, it only supports TCP, while large-scale deployments of Memcached use UDP for GET requests [12]. Like the CloudSuite 2 client, Mutilate [33] also only supports TCP. Furthermore, it does not have the ability to control item popularity. Memaslap [34] is a Memcached benchmarking tool that comes with the libmemcached library. Libmemcached is a C/C++ library that facilitates the development of clients for the Memcached server. Memaslap supports both TCP and UDP, but it only reports the minimum, maximum, mean, and standard deviation for latency, without reporting the quality-of-service percentage. Like Mutilate, Memslap does not have the ability to control item popularity.

VIII. Conclusions

With the increase in popularity of the cloud as a platform for delivering global-scale online services, it has become important to benchmark cloud workloads, to continue to improve the state of art of server systems. We attempted to use the existing cloud benchmarks, but found drawbacks in them – the most important one being that the benchmark design choices are not transparent to the end-user of the benchmark. In this work, we chronicled our experience of developing benchmarks for three network-intensive cloud applications and documented our design choices and rationale behind them. Specifically, we described three cloud applications: video-streaming, Web2.0, and object caching.

We compared our benchmarks with existing tools and demonstrated a number of distinct differences, concentrating on the aspects that have an impact on the results of the system-level measurements. In particular, we highlighted how the request mix and machine setup can have a significant impact on the performance of the cloud application under test. Finally, we showed that, despite major system-level performance differences, the micro-architectural behavior of the new benchmarks is similar to the CloudSuite 2 workloads.

Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1452904 and by a gift from Cavium, Inc. The experiments were conducted using equipment purchased through NSF CISE Research Infrastructure Grant No. 1513028.

We thank Dr. Tim Brecht and Jim Summers for their help in the development of the video-streaming benchmark and for providing us the source code from their prior work [16].

References

[1] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[2] "Googlecloudplatform – perfkitbenchmarker." [Online]. Available: https://github.com/GoogleCloudPlatform/PerfKitBenchmarker

[3] "The zettabyte era: Trends and analysis." [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html

[4] "Php: Hypertext preprocessor." [Online]. Available: http://php.net

[5] "Memcached - a distributed memory object caching system." [Online]. Available: http://memcached.org/

[6] "Docker - build, ship, and run any app, anywhere." [Online]. Available: https://www.docker.com

[7] "A network address translator (nat) traversal mechanism for media controlled by real-time streaming protocol (rtsp)." [Online]. Available: https://tools.ietf.org/html/draft-ietf-mmusic-rtsp-nat-22

[8] A. C. Begen, T. Akgul, and M. Baugher, "Watching video over the web: Part 1: Streaming protocols," *Internet Computing, IEEE*, vol. 15, no. 2, pp. 54–63, 2011.

[9] "Nginx – high performance load balancer, web server, and reverse proxy." [Online]. Available: https://www.nginx.com/

[10] "Olio – a web 2.0 toolkit." [Online]. Available: http://incubator.apache.org/projects/olio.html

[11] "Faban - helping measure performance." [Online]. Available: http://faban.org

[12] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook." in *nsdi*, vol. 13, 2013, pp. 385–398.

[13] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 36–47, 2013.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[15] "Elgg - open source social networking engine." [Online]. Available: https://elgg.org

[16] J. Summers, T. Brecht, D. Eager, and B. Wong, "Methodologies for generating http streaming video workloads to evaluate web server performance," in *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 2012, p. 2.

[17] D. Mosberger and T. Jin, "httperf – a tool for measuring web server performance," vol. 26, no. 3. ACM, 1998, pp. 31–37.

[18] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: a view from the edge," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 15–28.

[19] "Why netflix chose nginx as the heart of its cdn." [Online]. Available: https://www.nginx.com/blog/why-netflix-chose-nginx-as-the-heart-of-its-cdn

[20] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, "Comparing and evaluating epoll, select, and poll event mechanisms," in *Linux Symposium*, vol. 1, 2004.

[21] M. Carbone and L. Rizzo, "Dummynet revisited," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, pp. 12–20, 2010.

[22] "Recommended upload encoding settings." [Online]. Available: https://support.google.com/youtube/answer/1722171

[23] "Akamai's state of the internet: Q1 2015 report." [Online]. Available: https://www.stateoftheinternet.com/resources-connectivity-2015-q1-state-of-the-internet-report.html

[24] T. o'Reilly, *What is web 2.0*. " O'Reilly Media, Inc.", 2009.

[25] "Powered by elgg." [Online]. Available: https://elgg.org/powering.php

[26] "Alexa statistics for facebook." [Online]. Available: http://www.alexa.com/siteinfo/facebook.com

[27] "Scaling in the linux networking stack." [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

[28] "Introduction to intel ethernet flow director and memcached performance," 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf

[29] "Dockerhub." [Online]. Available: https://hub.docker.com/

[30] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy, "Benchlab: an open testbed for realistic benchmarking of web applications," in *Proceedings of the 2nd USENIX conference on Web application development*. USENIX Association, 2011.

[31] P. Pegus, E. Cecchet, and P. Shenoy, "Video benchlab: an open platform for realistic benchmarking of streaming media workloads," in *Proc. ACM Multimedia Systems Conference (MMSys), Portland, OR*, 2015.

[32] "Standard performance evaluation corporation." [Online]. Available: http://www.spec.org/web2009

[33] "Mutilate: high-performance memcached load generator." [Online]. Available: https://github.com/leverich/mutilate

[34] "Memaslap – an open source c/c++ client library and tools for the memcached server." [Online]. Available: http://libmemcached.org/libMemcached.html