Studying Hardware and Software Trade-Offs for a Real-Life Web 2.0 Workload

Stijn Polfliet Frederick Ryckbosch Lieven Eeckhout
ELIS Department, Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{stijn.polfliet, frederick.ryckbosch, lieven.eeckhout}@elis.UGent.be

ABSTRACT

Designing data centers for Web 2.0 social networking applications is a major challenge because of the large number of users, the large scale of the data centers, the distributed application base, and the cost sensitivity of a data center facility. Optimizing the data center for performance per dollar is far from trivial.

In this paper, we present a case study characterizing and evaluating hardware/software design choices for a real-life Web 2.0 workload. We sample the Web 2.0 workload both in space and in time to obtain a reduced workload that can be replayed, driven by input data captured from a real data center. The reduced workload captures the important services (and their interactions) and allows for evaluating how hardware choices affect end-user experience (as measured by response times).

We consider the Netlog workload, a popular and commercially deployed social networking site with a large user base, and we explore hardware trade-offs in terms of core count, clock frequency, traditional hard disks versus solid-state disks, etc., for the different servers, and we obtain several interesting insights. Further, we present two use cases illustrating how our characterization method can be used for guiding hardware purchasing decisions as well as software optimizations.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: Modeling of computer architecture; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*

General Terms

Design, Performance, Measurement, Experimentation

Keywords

Data center, Web 2.0, performance analysis

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

1. INTRODUCTION

Internet usage has grown by 480% over the past ten years worldwide according to a recent study by Internet World Stats[1]. This fast increase is due to various novel Internet services that are being offered, along with ubiquitous Internet access possibilities through various devices including mobile devices such as smartphones, tablets and netbooks. Online social networking in particular has been booming over the past few years, and has been attracting an increasing number of customers. Facebook, for example, has more than 800 million active users as of January 2012[2], and 50% of these users log on to Facebook at least once a day. Twitter generates 140 million tweet messages per day as of February 2011[3]. LinkedIn has more than 135 million professionals around the world as of November 2011[4]. Netlog, a social networking site where users can keep in touch with and extend their social network, is currently available in 40 languages and has more than 94 million users throughout Europe as of January 2012[5]. Clearly, social networking communities have become an important part of digital life.

Designing the servers and data centers to support social networking is challenging, for a number of reasons. As mentioned above, social networks have millions of users, which requires distributed applications running in large data centers [2]. The ensemble of servers is often referred to as a warehouse-scale computer [3] and scaling out to this large a scale clearly is a major design challenge. Because of their scale, data centers are very much cost driven — optimizing the cost per server even by only a couple tens of dollars results in substantial cost savings and proportional increases in profit. There are various factors affecting the cost of a data center, such as the hardware infrastructure (servers, racks and switches), power and cooling infrastructure, operating expenditure, and real estate. Hence, data centers are very cost-sensitive and need to be optimized for the ensemble. As a result, operators drive their data center design decisions towards a sweet spot that optimizes performance per dollar.

A key question when installing a new data center obviously is which new hardware infrastructure, i.e., which servers, to buy. This is a non-trivial question given the many constraints. On the one hand, the hardware should be a good fit for the workloads that are going to run in the data center. The workloads themselves

[1] http://internetworldstats.com/stats.htm
[2] http://www.facebook.com/press/info.php?statistics
[3] http://blog.kissmetrics.com/twitter-statistics/
[4] http://press.linkedin.com/about
[5] http://en.netlog.com/go/about

could be very diverse — some workloads are interactive, others are batch-style workloads and thus throughput-sensitive and not latency-critical; some workloads are memory-intensive while others are primarily compute-intensive or I/O-intensive. Hence, some compromise middle-of-the-road architecture may need to be chosen to satisfy the opposing demands; alternatively, one may opt for a heterogeneous system where different workloads run on different types of hardware. Further, one needs to anticipate what new workloads might emerge in the coming years, and how existing workloads are likely to evolve over time. On the other hand, given how cost-sensitive a data center is, it is of utmost importance that the correct hardware is purchased for the correct task. High-end hardware is expensive and consumes significant amounts of power, which leads to a substantial total cost of ownership. This may be the correct choice if the workloads need this high level of performance. If not, less expensive and less power-hungry hardware may be a much better choice.

It is exactly this purchasing question that motivated this work: Can we come up with a way of guiding service operators and owners of data centers to what hardware to purchase for a given workload? Although this might be a simple question to answer when considering a single workload that runs on a single server, answering this question is quite complicated when it comes to a Web 2.0 social networking workload. A social networking workload consists of multiple services that run on multiple servers in a distributed way in a data center, e.g., Web servers, database servers, memcached servers, etc. The fundamental difficulty that a Web 2.0 workload imposes is that the performance of the ensemble can only be measured by modeling and evaluating the ensemble, because of the complex interplay between the various servers and services. In other words, performance as perceived by the end-user, i.e., the response times observed by the end user, is a result of the performance of the individual servers as well as the overall interaction among the servers. Put differently, optimizing the performance of an individual server may not necessarily be beneficial for the ensemble and may not necessarily have impact on end-user experience, nor may it have impact on the total cost of ownership.

In this paper, we present a case study in which we characterize a real-life Web 2.0 workload and evaluate hardware and software design choices. We sample the Web 2.0 workload both in space and in time to obtain a reduced workload that can be replayed, driven by real input data. The reduced workload captures the important services (and their interactions) and allows for evaluating how hardware choices affect end-user experience.

We consider Netlog's commercially used Web 2.0 social networking workload, and we evaluate how hardware design choices such as number of cores, CPU clock frequency, hard-disk drive (HDD) versus solid-state drive (SSD), etc. affect overall end-user perceived performance. We conclude that the number of cores per node is not important for the Web servers in our workload, hence the hardware choice should be driven by cost per core; further, we find that the end-user response time is inversely proportional to Web server CPU frequency. SSDs reduce the longest response times by around 30% over HDDs in the database servers, which may or may not be justifiable given the significantly higher cost for SSD compared to HDD. Finally, the memcached servers show low levels of CPU utilization while being memory-bound, hence the hardware choice should be driven by the cost of integrating more main memory in the server.

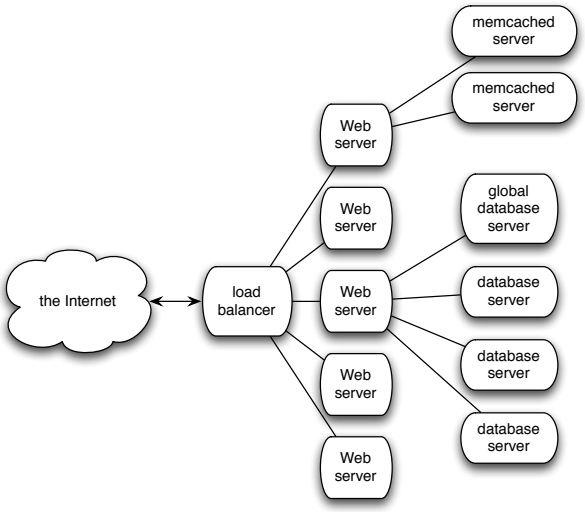

Figure 1: Netlog's architecture.

We believe that this approach is not only useful to service providers and data center owners, but also to architects, system builders, and integrators to understand Web 2.0 workloads and how hardware choices affect user-perceived performance, server throughput, and utilization. Further, software developers and data center system administrators may find the approach useful to identify and solve performance bottlenecks in the software and experiment with alternative software implementations. To demonstrate the potential usage of the characterization, we present two uses cases illustrating how it can be leveraged for guiding hardware purchasing decisions and software optimizations.

This paper is organized as follows. We first describe the Web 2.0 workload used in this study (Section 2). We then set the goals for this paper (Section 3) and describe our methodology in more detail (Section 4). We detail our experimental setup (Section 5) and then present our results (Section 6). We focus on two important use cases for this work (Section 7). Finally, we discuss related work (Section 8) and conclude (Section 9).

2. NETLOG WEB 2.0 WORKLOAD

As mentioned in the introduction, we use Netlog's software infrastructure as a representative Web 2.0 workload. Netlog hosts a social networking site that is targeted at bringing people together. As of January 2012, Netlog is currently available in 40 languages and has more than 94 million members throughout Europe. According to ComScore[6], Netlog is the pageview market leader in Belgium, Italy, Austria, Switserland, Romania and Turkey; and it is the second market leader in the Netherlands, Germany, France and Portugal. Netlog has around 100 million viewers per month, leading to over two billion pageviews per month. Netlog users can chat with other friends, share pictures, write blog entries, watch movies and listen to music.

Netlog's architecture is illustrated in Figure 1. A load balancer

[6]http://www.comscore.com/

distributes the incoming requests among the Web servers. The Web servers process the requests and assemble a response by fetching recently accessed data from the memcached servers. If the requested data is not present in one of the memcached servers, the Web server communicates with one of the database servers. There is one global database that holds general information with user data (like nickname and passwords). All other user data is spread among multiple database servers using a technique called 'sharding'[7]. Each of the servers run on a physical machine. The relative fraction of servers is as follows: 54% of Netlog's servers are Web servers, 16% are memcached servers and 30% are database servers. The Netlog data center hosts more than 1,500 servers.

Netlog's data center is partitioned among the languages that it supports, i.e., servers are devoted to one particular language. The largest language is Dutch, followed by German, Italian, Arabic, English, and others. Interestingly, usage patterns are similar across languages, hence, the same relative occurrence of Web, caching and database servers is maintained across all the languages.

In terms of software, the Web servers run the Apache HTTP server[8]; the caching servers run Memcached[9]; and the database servers run MySQL[10]. For more information, please refer to Section 5.

3. CASE STUDY GOALS

Before describing our case study in great detail, we first need to set out its goals. First, we want to be able to characterize and evaluate end-user perceived performance of a Web 2.0 system. This implies that a representative part of the workload needs to be duplicated in the experimental environment which enables evaluating overall end-to-end performance. This in turn implies that a set of machines needs to be engaged with each machine running part of the workload — some run Web servers, some run database servers, others run memcached servers. Collectively, this set of machines runs the entire workload. This experimental environment, when supplied with real user requests, will act like a real data center running the real workload. This enables measuring user-perceived response times as well as server-side throughput and utilization.

Second, the experimental environment by itself will not provide useful measurement data. It also needs a method to feed real-life user requests into the experimental environment. In other words, real user requests need to be captured and recorded in a real data center and then need to be replayed in our experimental environment. This will enable us to measure how design choices in the hardware and the software affect user-perceived performance as well as server throughput and utilization.

Third, in addition to being able to faithfully replay real-life user requests, it is useful to be able to stress the setup through experiments in which user requests are submitted at a fixed rate. This allows for gaining insight into the system's limits and how the system will react in case of high loads. For example, it allows for learning about how user-perceived response time is affected by server load. Or, it allows for understanding the maximum allowable server load before seeing degradations in user response times.

Finally, we need the ability to run reproducible experiments, or in other words, we want to draw similar performance figures when running the same experiment multiple times. This allow us to measure how changes in system configuration parameters affect performance. In the end, we want to use the experimental environment and change both hardware and software settings to understand how hardware and software design choices affect user-perceived performance as well as server-level throughput and utilization. This not only enables service providers and data center owners to purchase, provision and configure their hardware and software, it also enables architects, system builders and integrators, software developers, etc. where to focus when optimizing overall system performance.

4. METHODOLOGY

Our methodology has a number of important features in order to make the experimental environment both efficient and effective for carrying out our case study.

- **Sampling in space.** It is obviously prohibitively costly to duplicate an entire Web 2.0 workload with possibly hundreds, if not thousands, of servers in the experimental environment. We therefore sample the workload in space and we select a reduced but representative portion of the workload as the basis for the experimental framework. For the Netlog workload, we select one language out of the many languages that Netlog's workload supports; this language is representative for the other languages and for the Netlog workload at large. Sampling in space allows us to evaluate a commercial Web 2.0 workload with hundreds of servers in real operation with only 10 servers in our experimental environment.

- **Sampling in time.** Replaying a Web 2.0 workload using real-life user input, as we will describe next, can be very time-consuming, especially if one wants to replay multiple days of real-life operation in the data center. Moreover, in order to understand performance trends across hardware and software design changes, one may need to explore many configurations and hence run the workload multiple times. This may make the experimental setup impractical to use. Hence, we analyze the time-varying behavior of the workload and we identify representative phases in the execution, which we sample from, and which we can accurately extrapolate performance numbers to the entire workload. Sampling in time allows to analyze only a few hours of real time while being representative for a workload that runs for days.

- **Workload warm-up.** Sampling in time implies that we evaluate only a small fraction of the total run time. A potential pitfall with this approach is that system state might be very different when replaying under sampling than if one were to replay a workload for days of execution. This is referred to as the cold-start problem. In other words, the system needs to be warmed up when employing sampling in time

[7]Sharding is a horizontal partitioning database design principle whereby rows of a database table are held separately, rather than splitting by columns. Each partition forms part of a shard, which may in turn be located on a separate database server or physical location.

[8]http://httpd.apache.org/

[9]http://memcached.org/

[10]http://www.mysql.com/

so that the performance characteristics during the evaluation are representative for as if we were to run the entire workload. Our methodology uses a statistics-based approach to gauge whether the system is warmed up sufficiently.

- **Replaying empirical user request streams.** As mentioned before, we capture and replay real-life user requests. The user request file that we store on disk and that we use as input to the experimental environment contains sufficient information for faithfully replaying real-life users requests. In other words, the input served to the load balancer of the Netlog workload is identical under replay as when we captured it during real-life operation.

We now discuss the various steps of our methodology in more detail.

4.1 Sampling in space

As part of this study we duplicated Netlog's workload. Because it is infeasible to duplicate Netlog's entire workload, we chose to duplicate a small part only, namely the part associated with the Slovene language. This is feasible to do, and leads to a representative workload. Netlog organizes its servers such that there are a number of physical servers per language domain. Hence, by selecting a language domain and by only duplicating that language domain, we sample in space while being representative for the entire workload. The Slovene part is representative for Netlog's entire workload because it exhibits the same partitioning of servers as the rest of Netlog's workload. Also, we observe similar degree of activity and access behavior (access to profiles, photos, videos, etc.) for the Slovene language as for the other languages.

Duplicating the Slovene language part of Netlog's workload can be done with a reasonable number of servers. Our setup includes 6 Web servers, 1 memcached server and 2 database servers; this distribution across server types is identical to what is observed for the entire Netlog workload, across all the languages. Further, our setup includes the entire Slovene database and all of its records. The data present in our duplicate copy is anonymized. This is done through hashing while maintaining the length of the records.

4.2 Validating the setup

Duplicating a Web 2.0 workload is a significant effort and involves fine-tuning various software settings and configurations that necessitates proper validation. We validated our experimental framework both functionally and with respect to behavior and timing. In particular, we automatically verified whether the file sizes returned by the duplicated workload match the file sizes observed in the real workload. The reason for doing so is that some of the Web pages returned by the Web 2.0 workload are composed semi-randomly, and hence its content may not be perfectly identical when requesting the same page multiple times. In our experimental environment, we found that 99.3% of the responses fall within a 5% error bound with respect to file size compared to the real workload environment, as shown in Figure 2.

4.3 Replaying user requests

An important aspect of the experimental environment is the replay of user requests. In order to do so, we collect user requests

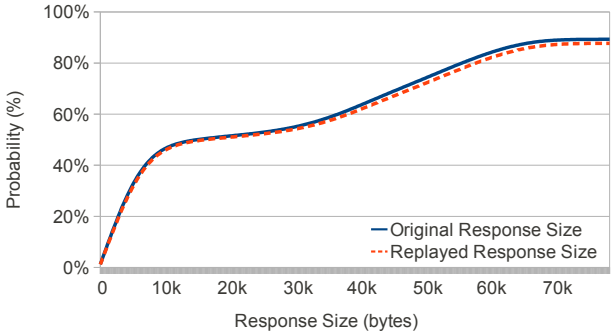

Figure 2: Distribution of response sizes when comparing real versus replayed requests.

as observed at the load balancer. The information collected by the user input recorder consists of the following items — recall that the data is anonymized:

- **Header information**. All HTTP header information is recorded so the same request can be reconstructed. This includes the requested URL, browser information, supported encoding formats, etc.

- **Timing information**. The date and time the request was submitted is recorded (at microsecond resolution). This allows for maintaining precise timing information when replaying the user request file. This is important to model bursty behavior in user requests.

- **User data**. The input recorder captures all POST data that is sent to the Web servers. Note that GET data is already captured as part of the URL in the header. All HTTP cookies are saved as well, and are used to do automated login.

The file that contains these user requests is fairly large and contains 24 GB of data per day on average. Our user input recorder uses `tcpdump`[11] to log the network traffic to a file in `pcap`[12] format. `pcap` defines an API for capturing network traffic. On Linux/Unix systems, this is implemented in the `libpcap` library which most network tools like tcpdump, Wireshark, etc. implement. A limitation of `tcpdump/pcap` is that it may drop packets; however, packet loss rate was less than 0.002% for a 1 Gbps network in our setup.

The replayer reads the user request file and replays the requests one by one. This means that the replayer picks the first request, sends the request to the Web 2.0 workload at the time specified in the request file. It then picks the next request and sends it at its time, etc. The replayer does not wait for the response to come back to determine the next request; all the requests are available in the user request file.

Implementing the user request replayer is a challenge in itself. The reason is that the user request file is huge in size, and the requests need to be submitted to the workload at a fine time granularity. Reading the request file from disk, and submitting requests in real-time is too slow. On the other hand, it is impractical to store

[11] http://www.tcpdump.org/

[12] http://www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html

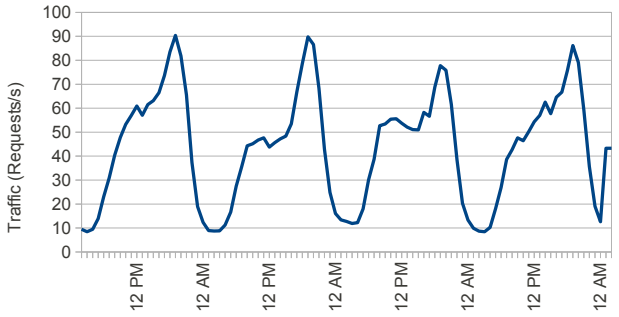

Figure 3: Netlog traffic profile for four days to the Slovene language domain.

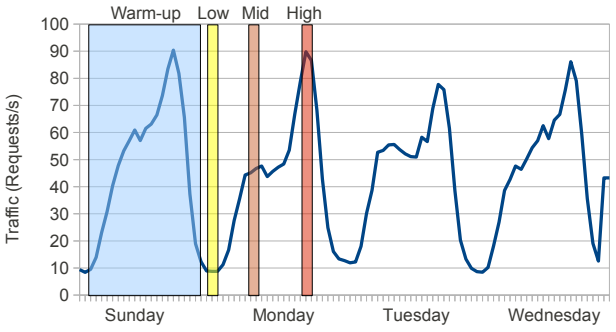

Figure 4: Identifying representative samples based on traffic intensity.

the entire request file in main memory. We therefore developed a two-thread replayer. The first thread reads the `pcap` file and fills in the requests in the request pool in memory. The second thread then reads from the request pool and submits the requests to the workload using `libcurl`[13], which is a client-side URL transfer library that supports sending requests using the HTTP protocol to a remote Web server.

4.4 Sampling in time

We recorded four days (March 13–16, 2011) of user activity to the Slovene language domain of the Netlog workload. This was done by capturing all the user requests (and their timing) at the load balancer. Replaying these four days of activity in real time would require four days of experimentation time. Although this is doable if one were to evaluate a single design point, exploring trade-offs by varying hardware and/or software parameters, quickly leads to impractically long experimentation times. We therefore employ sampling in time to evaluate only parts of the workload activity while being representative for the entire workload.

Figure 3 shows traffic over a four day period in number of requests per second. Clearly, we observe cyclic behavior in which there is much more activity in the evening than during the day. Traffic increases steeply in the morning between 6am and 9am, and remains somewhat stable or increases more slowly between 9 am and 5pm. Once past 5pm, traffic increases steeply until 8pm. We observe a sharp decrease in the number of requests past 9pm. This traffic pattern suggests that sampling in time is a sensible idea, i.e., by picking samples that represent different traffic patterns, one can significantly reduce the load that needs to be replayed, which will lead to significant improvements in experimentation speed, while reproducing a representative workload.

We set ourselves a number of goals for how to sample in time. We want the samples to be representative in a number of ways: we want the samples to represent diverse traffic intensity as well as the sort of activity that the samples cover, i.e., as mentioned before, Netlog offers various sorts of services ranging from chatting to watching videos, etc., hence the samples should cover these different types of activity well. Further, we prefer having a few long representative samples over having many small samples. The reason is that small samples require more precise warmup of the system than longer samples in order to be accurate.

We therefore employ the following two-step sampling procedure. We first aim at finding a number of time periods with different traffic intensity. We employ k-means clustering as our classification method [5]. The input to the clustering algorithm is a time series representing the number of requests per minute. The clustering algorithm then aims at classifying this time series in a number of clusters N. It initially picks N cluster centroids in a random fashion, and assigns all data elements in the time series to its closest cluster. In the next iteration, the algorithm recomputes the cluster centroid, and subsequently reassigns all data elements to clusters. This iterative process is repeated until convergence, or until a maximum number of iterations is done. An important question is how many clusters N should one pick. We use the Bayesian Information Criterion (BIC) [9], which is a measure for how well the clustering matches the data. Using a maximum value of $N_{max} = 6$ — recall we aim for a limited number of samples — we obtain the result that $N = 3$ yields the optimum BIC score. Hence, we obtain three samples. These are shown in Figure 4. Intuitively, these three samples correspond to low-intensity, medium-intensity and high-intensity traffic, respectively.

The next question is how long the samples should be in these low, medium and high-intensity traffic regions. We therefore rely on our second requirement: we want the samples to cover diverse behavior in terms of the type of traffic. We identify 30 major types of traffic including messages, photos, videos, friends, music, etc. This yields a 30-dimensional time series: each data element in the time series consists of 30 values, namely the number of requests per minute for each type of traffic. We then apply k-means clustering on this 30-dimensional time series which yields the optimum number of four clusters using the BIC score. These four clusters represent the predominant traffic rates observed at a given point in time. Figure 5 illustrates how the time series of ten hours of the second day is distributed across these four clusters. Interestingly, some traffic rates are more predominant during some periods of time, and traffic rate predominance varies fairly quickly. However, if we take a long enough snapshot, e.g., two hours, the sample contains all traffic rates. The end result for sampling in time, thus is that we pick three samples of two hours of activity from the low, medium and high-intensity regions.

4.5 Warmup

With sampling in time, an important issue is how to start from a warmed-up system state so that the performance numbers that we obtain from our experiments are representative for the real work-

[13] http://curl.haxx.se/

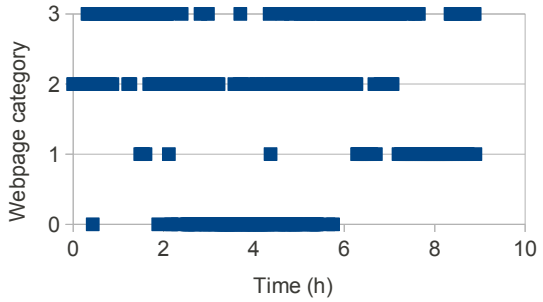

Figure 5: Traffic classified by its type.

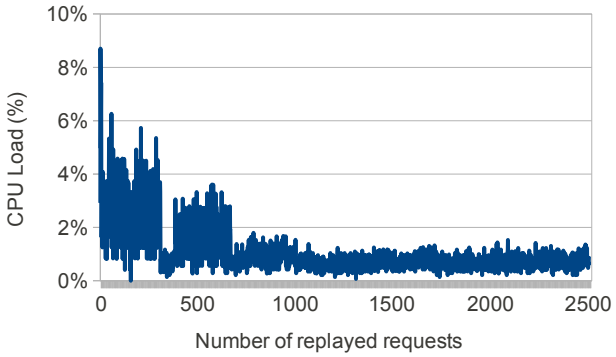

Figure 6: Quantifying PHP cache warmup behavior. Replay speed is set to a fixed rate of 10 requests/s.

load. Clearly, starting from a cold state is not going to be accurate because the performance of the workload will be very different from what one would observe in a real (and warmed-up) environment. Warmup of a Web 2.0 workload involves a number of issues. First, as mentioned before, the Web servers run PHP code, and hence they rely on an opcode cache that caches the bytecodes; the PHP engine does not need to interpret cached bytecodes again, and hence it achieves better performance. This implies that the performance of the PHP engine is relatively low initially, but then improves gradually as more and more code gets cached and optimized; this is obviously reflected in the Web server response times observed by the end user. In other words, in the context of this work, it is important that we measure the performance of the PHP engine in steady-state modus, in which it executes highly optimized code as opposed to interpreting the PHP code. As shown in Figure 6, the CPU load is higher when the PHP engine is first initialized. In this stage, the PHP engine still has to compile all PHP code. After 1,000 requests most PHP pages are compiled and loaded into the cache, hence, we conclude that the PHP cache is warmed up in the order of a couple seconds.

Second, and more importantly, we also need to warm up the memcached and database servers. Initially, in a cold system, all the requests will go to the database server because the memcached server does not cache any data yet; further, the database server will need to read from disk to access the database. Hence, we will observe a significant fraction of time spent waiting for I/O both over the network and for accessing disks. Indeed, gigabytes of data need to be read in the database and transferred from the database servers to a memcached server. This requires a large number of user requests being sent to the system to warmup the database and memcached servers. Figure 7 illustrates the fraction I/O wait time on the database server starting from a cold state as a function of time. We observe that the fraction I/O wait time, which is proportional to how often one needs to access the database on disk and transfer data to the memcached server, decreases as a function of time. Although there is a steep decrease in I/O wait time in the first few hours, it takes close to an entire day before I/O wait time drops below a few percent which represents a fully warmed up system.

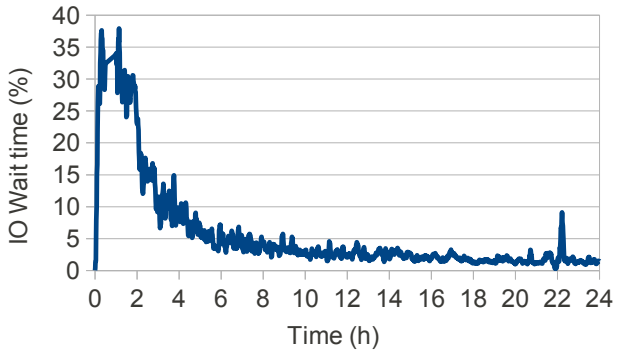

Figure 7: Quantifying how long one needs to warmup the database and memcached servers: I/O wait time on the database server is shown as a function of time when replaying the first day.

In order to get more confidence in this finding we employ the Kolmogorov-Smirnov statistical test to verify whether the system is sufficiently warmed up. The Kolmogorov-Smirnov test is a non-parametric test for the equality of continuous, one-dimensional probability distributions. It basically measures whether two distributions are equal or not; the exact form of the distribution is not important, hence it is labeled a non-parametric test. In this work, we compare the distribution of user response times starting from a cold versus a warmed-up system. This is done in steps of 5,000 user requests, see Figure 8. The P-value reported by the Kolmogorov-Smirnov test gives an estimate for how good the correspondence is between starting from no-warmup versus a fully warmed up system; the P-value is a higher-is-better metric. We observe that the P-value saturates after approximately six hours of warmup, and reaches its highest score after 18 to 20 hours of warmup. Based on these observations we decided to warm up our experimental system with one full day of load.

Note that, in our experimental environment, it does not take a full day to actually warmup the entire system. During warmup, we quickly submit an entire day's user requests to the Netlog workload, as fast as possible. This takes approximately two hours in our setup. Once the system is warmed up, we then submit user requests for the sample of interest at the time stamps as stored in the user request file, as explained before.

5. EXPERIMENTAL SETUP

As mentioned before, we duplicated the Slovene language domain of the Netlog workload to our experimental environment. Our infrastructure consists of 10 dual AMD Opteron 6168 servers, with each server having 24 cores in total or 12 cores per CPU. Each

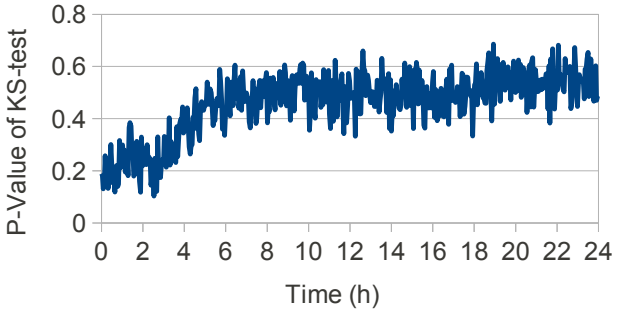

Figure 8: Using the Kolmogorov-Smirnov test to verify whether the system is sufficiently warmed up by comparing the distribution of response times under full versus no warmup.

server has at least 64 GB of main memory, and is equipped with both a regular HDD (1 TB Seagate SATA 7200 rpm) as well as an SSD (128 GB ATP Velocity MII). We configure the machines as 6 Web servers, 1 memcached server, and 2 database servers. The tenth server is used to generate workload traffic and inject user requests to the system under test.

Our baseline configuration runs all the cores at 1.9 GHz. We provision the Web servers as well as the database servers with 64 GB of main memory. The memcached server is equipped with 128 GB of RAM. Further, we assume a HDD drive in each of the servers — we consider SSD in the database servers in one of the experiments.

Our infrastructure uses Ubuntu 10.04[14]. The Web server is configured with Apache 2.2[15] and runs PHP 5.2[16]. The database software used is a MySQL derivative, Percona 5.1. We use the standard Memcached 1.4.2[17] version as our caching mechanism.

6. RESULTS AND DISCUSSION

Using our experimental environment, we now focus on gaining insights in how hardware trade-offs affect user-perceived performance (response times) for the end-to-end workload. We first consider user requests submitted at the rate as measured in the real-life workload, and we look at hardware trade-offs for the Web server, memcached server and database server, respectively. Subsequently, we consider fixed-rate experiments in order to stress the system.

6.1 Web server

We evaluate two hardware trade-offs for the Web server, namely CPU clock frequency and the number of cores per node. Figure 9 shows the distribution of the user response times while changing the Web server's CPU frequency in three steps: 1.9 GHz, 1.3 GHz and 800 MHz. The distribution of response times is skewed, i.e., there is a peak in the response time distribution around 0.04 seconds at 1.9 GHz, and the distribution has a fairly long and heavy tail for longer response times. We observe similarly skewed distributions at lower CPU clock frequencies, yet the distributions shift towards the right with decreasing frequencies, i.e., user response time increases with lower clock frequencies. This is perhaps intuitive, as

[14]http://www.ubuntu.com
[15]http://www.apache.org/
[16]http://www.php.net
[17]http://www.memcached.org/

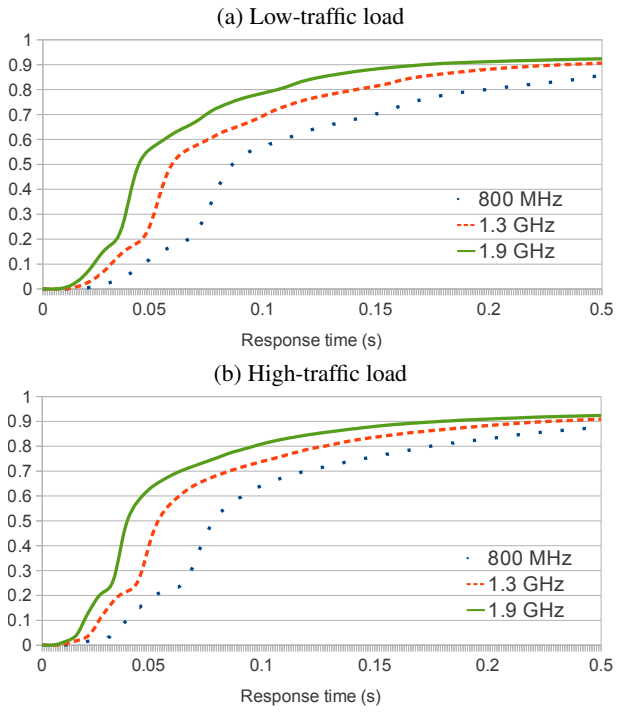

Figure 9: Distribution of user response times while changing the Web server's CPU frequency under (a) low-traffic load and (b) high-traffic load.

the CPU gets more work done per unit of time at higher clock frequencies. It is interesting to observe though that Web server clock frequency has a significant impact on user response times (even at low CPU loads, as we will see next). In conclusion, user response time is sensitive to Web server clock frequency. Hence, the Web server should have a sufficiently high clock frequency in order not to exceed particular bounds on user response time.

An important point of concern in provisioning servers for a Web 2.0 workload is to have sufficient leeway to accommodate bursty traffic behavior and sudden high peaks of load. For gauging the amount of leeway on a server, we use CPU load. If the CPU load is sufficiently low, this means that the server can accommodate additional work. Figure 10 quantifies Web server CPU load as a function of clock frequency. Clearly, CPU load increases with lower CPU frequencies.

As alluded to before, the response time distribution has a fairly long and heavy tail. A heavy-tailed response time distribution is a significant issue in Web 2.0 workloads because it implies that some users are experiencing an unusually long response time. Given the large number of concurrent users of Web 2.0 workloads, and although the number is small in terms of percentages, still a significant number of users will be experiencing very long response times. Very long and unpredictable response times quickly irritate end users, which may have a significant impact on company revenue if users sign off because of the slow response times. Because of this, companies such as Google and others heavily focus on the 99% percentile of the user response times when optimizing overall system performance. Figure 11 shows the percentile response times as a function of Web server CPU clock frequency. For exam-

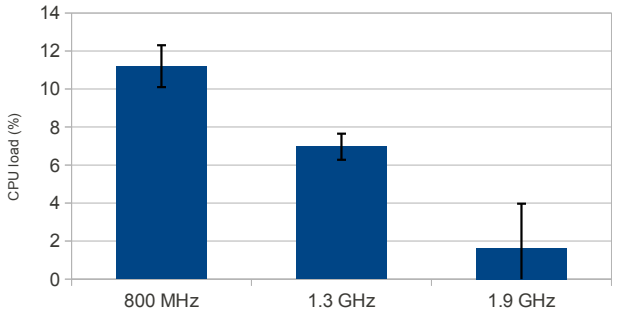

Figure 10: Web server CPU load as a function of CPU clock frequency for the high-traffic load scenario.

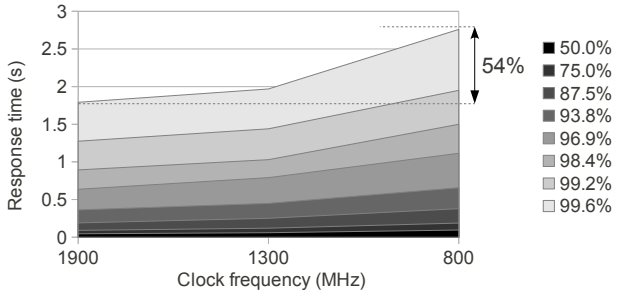

Figure 11: Percentile response times as a function of Web server CPU clock frequency.

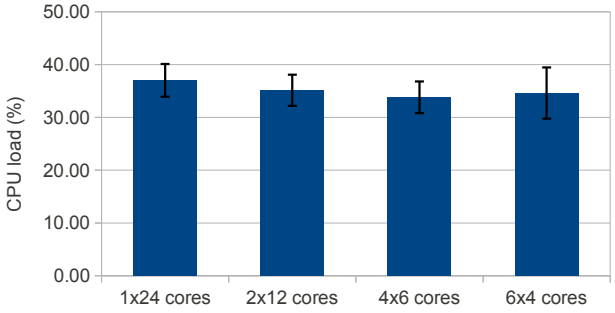

Figure 12: Web server CPU load as a function of the number of nodes and cores per node: $m \times n$ means m nodes and n cores per node.

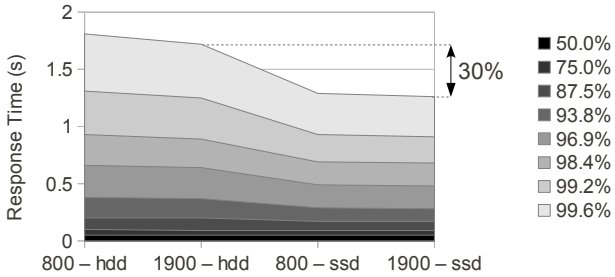

Figure 13: Trading off HDD versus SSD and CPU clock frequency for the database servers.

ple, this graph shows that, see the top left, 99.6% of the response times are below 1.8 seconds at 1.9GHz. The 99.6% percentile goes up to 2.8 seconds at 800MHz, see the top right. The interesting observation is that long-latency response times increase sub-linearly with decreasing Web server clock frequency. The 99.6% percentile response time increases by 54% only, while decreasing clock frequency from 1.9GHz to 800MHz, or increasing cycle time by 138% from 0.53ns to 1.25ns.

The second hardware trade-off that we study relates to the number of cores per node one should have for the Web server. The reason why this is an interesting trade-off is that systems with more sockets per node are more expensive, i.e., a four-socket system is typically more than twice as expensive as a two-socket system. Similarly, the number of cores per CPU also directly relates to cost. Figure 12 quantifies Web server CPU load as a function of the number of nodes and the number of cores per node. (Recall that CPU load is a good proxy for user response time as observed.) We vary from one Web server node with 24 cores enabled, to 2 nodes and 12 cores each, to 4 nodes and 6 cores each, to 6 nodes and 4 cores each. Clearly, CPU load (and response time) is not affected much by node and core count (as long as the total number of cores is constant). This suggests that the Web server is a workload that scales well with core count, even across nodes. In conclusion, when purchasing Web server hardware, although total core count is important, core count per node is not. This is an important insight to take into account when determining how many servers to buy with how many cores each. Determining the best buy (number of servers and number of cores per server) depends on many factors such as performance, power cost, real estate cost, reliability, availability, etc., however, this case study shows that the number of cores per server is a parameter one can tweak to optimize Web server performance per dollar.

6.2 Database server

As mentioned before, the database servers generate substantial disk I/O activity. We therefore focus on a hardware trade-off that involves HDDs versus SSDs in the database servers. We also vary CPU clock frequency. Figure 13 quantifies the percentile response times for the four hardware design points that result from changing clock frequency and hard drives. We observe that, while short response times are not greatly affected by replacing the HDD with an SSD, the 99.6% percentile response time decreases by 30% when trading an HDD for an SSD. Although this is a significant reduction in the longest response times observed, it may not justify the significantly higher cost of SSD versus HDD.

6.3 Memcached server

The memcached server has a very low typical CPU load, and is primarily memory and network-bound. The average CPU load for the memcached server is typically below 5% when stressed with 6 Web servers. Figure 14 shows CPU time versus network time for a memcached experiment in which we generate memcached GET requests of varying size, more specifically, the responses of the GET requests are of varying size. This clearly shows that memcached performance is mainly determined by the network. Hence, CPU performance for the memcached servers is not critical, and one could for example deploy relatively inexpensive servers. It is

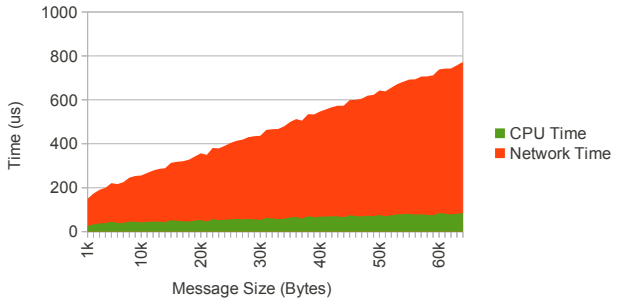

Figure 14: CPU time versus network time for memcached requests of different size.

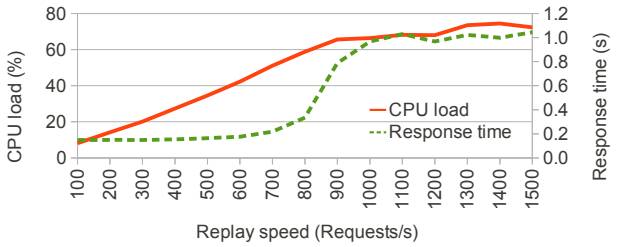

Figure 15: CPU load and average response time as a function of the number of requests per second under a fixed-rate experiment.

important for the memcached servers to have sufficient amount of memory though.

6.4 Fixed-rate experiments

The experiments done so far involved replaying the user requests as recorded in the real-life workload, i.e., the requests are submitted at a rate determined by the users. We now consider experiments in which we submit requests at a fixed rate. The reason for doing so is to stress the system under high levels of request rates in order to understand how CPU load and user response times are affected by the load imposed on the entire workload. Figure 15 quantifies CPU load of the Web servers (left axis) and the average response time (right axis) as a function of the number of requests per second submitted to the system. Interestingly, the response time remains low and CPU load increases linearly with request rate, up until a request rate of 700 requests per second. Beyond 800 requests per second, CPU load saturates around 65 to 75 percent, and response time increases substantially from 0.2 seconds to approximately 1 second. The reason why response time saturates beyond 1,000 requests per second is that requests get dropped once the Web server's CPU load gets too high as it runs out of resources and is unable to process all incoming requests. Note that handling a request is more than just generating static content: every Web server request typically initiates several memcached and database requests.

7. USE CASES

The approach we have followed can be applied to numerous use cases. For example, data center owners and service providers can use the approach to guide purchasing decisions. Similarly, system architects, integrators and implementors can use the approach to gain insight in how fundamental design decisions of the data center architecture affect user perceived performance. Finally, software architects and system administrators can use the approach to drive software design decisions, identify and address performance bottlenecks and evaluate alternative software implementations.

In this section, we present two use cases to illustrate the potential of the approach for making hardware and software design choices.

7.1 Hardware purchasing

The first use case that we present relates to hardware design choices, and more specifically to data center owners and service providers who wish to understand which hardware to purchase for a given workload. As mentioned earlier, this is a challenging question because of the many constraints one needs to deal with, ranging from purchasing cost, energy/power cost, cooling cost, performance, throughput, density, etc. In this use case, we look at two of the most important factors, namely performance and purchasing cost, for guiding the purchasing decisions. We also consider the implications on a third factor, namely power consumption.

Data center owners and service providers who wish to upgrade their hardware infrastructure face a challenging problem, and their decisions are mostly guided by experience and advice given by the hardware vendor(s). We now describe a scenario in which a hardware vendor would make a recommendation on which hardware to purchase. This scenario is hypothetical — we did not actually ask a hardware vendor for making a suggestion for a specific configuration for the given Web 2.0 workload. However, the suggested configuration is based on rules of thumb, and therefore we believe it is realistic. The hardware prices are based on real cost numbers of a large online hardware vendor. We now describe the suggested hardware configuration.

- **Web server**: It is well-known that Web servers are performance-hungry. Therefore, a hardware vendor might, for example, suggest a high-performance system with an Intel Xeon X3480 (3.06 GHz, 8MB LLC Cache, 4 cores, Hyper-Threading), 8 GB RAM and a typical HDD. The price for this web server is $1,795.

- **Memcached server**: Because memory is an important factor in a memcached server, the vendor might suggest including more memory, leading to an Intel Xeon X3480, with 16 GB RAM and a typical HDD. The price for this system is $2,015.

- **Database server**: Finally, because the hard disk is often a bottleneck on a database server, a hardware vendor might suggest to replace the HDD with an SSD, leading to a system with an Intel Xeon X3480, 16 GB RAM and an SSD. The price for this database server is $2,915.

The total cost of this configuration, including 6 Web servers, 1 memcached server and 2 database servers — recall the 6-1-2 ratio of web, memcached and database servers in the Netlog configuration as described earlier — equals $18,615.

Now, given the insight obtained from this study as described in Section 6, we can make the following alternative recommendations for a hardware configuration.

- **Suggestion #1: Low-cost memcached and database server**

  As previously reported, a memcached server does not need a high-performance CPU. We therefore suggest a CPU of

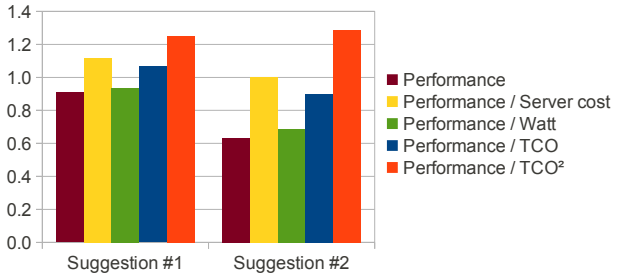

Figure 16: Several performance trade-offs for different hardware suggestions compared to the hardware vendor suggestion.

the same class as proposed by the hardware vendor, but at a lower clock frequency (e.g., Intel Xeon X3440 at 2.53 GHz). The same is true for the database server. On top of that we suggest not to consider an SSD, because of its high cost and relatively low performance gain over HDD for this particular workload.

The lower price for the CPU makes the memcached and database server cost $1,445 each. The total price of our suggested configuration now equals $15,105. This means a purchasing cost reduction of 18.9%.

Using the results presented in Figure 13, we conclude that, using this configuration, 50% of all requests will not experience any extra latency. For the other 50% of the requests, response times would increase from 11% for the 75% percentile to 39% for the 99.6% percentile. In summary, performance as perceived by the end user would be reduced by 9.1% on average. It is then up to the service provider to balance the purchase cost against the loss in performance for a small fraction of the user requests.

- **Suggestion #2: Low-frequency Web server**

  We can go one step further and use a CPU at lower clock frequency for the Web servers as well (Intel Xeon X3440 at 2.53 GHz). The price of the Web server is now $1,225. This could mean a total purchasing cost reduction of 37.2% over the hardware vendor suggested configuration.

  User requests would now observe a latency increase by 29% for the 50% percentile, and up to 56% for the 99.6% percentile. On average, end-user performance would be reduced by 36.9%. Again, it is up to the service provider to determine whether this loss in performance is worth the reduction in cost.

So far, when computing the cost reduction, we only focused on purchasing cost and we did not account for savings due to lower power consumption, leading to reduction in cost for powering and cooling the servers. Power consumption is a significant cost factor in today's servers and data centers [3], hence it should be taken into account when computing cost savings. In Figure 16, we show different metrics to help the service provider determine which platform should be chosen; the reason for considering multiple metrics is that different criteria might be appropriate for different scenarios. All metrics are higher-is-better metrics, and all values are normalized against the suggestion by the hardware vendor; a value greater than one thus is in favor of one of our suggestions.

- **Performance:** As mentioned before, raw performance drops by 9.1% for suggestion #1 and 36.9% for suggestion #2. This metric does not take any cost factor into account.

- **Performance per server cost:** Suggestion #1 reduces cost without dramatically reducing performance. When using server hardware costs in our metric, the benefit for suggestion #1 is 12.0%. The benefit for suggestion #2 is almost zero because of the extra decrease in performance, i.e., cost saving is offset by performance decrease.

- **Performance per Watt:** In the above case study, we considered two server configurations, one at 3.06 GHz and one at 2.53 GHz, which corresponds to a 17.3% reduction in clock frequency. Dynamic power consumption is proportional to clock frequency, so CPU dynamic power consumption will be lowered by 17.3% as well. The Intel X3480 has a Thermal Design Point (TDP) of 95 Watts and we assume other components (motherboard, disk, memory, etc.) to consume 100 Watts in total. The reduce in wattage is low compared to the performance decrease, resulting in a net decrease in performance per Watt for the two suggestions compared to the hardware vendor's suggestion. However, this metric only considers power consumption and does not take electricity costs into account.

- **Performance per TCO:** As mentioned before, data center facilities and online services are cost-sensitive, and hence, a metric for the data center should include some notion of total cost of ownership (TCO). TCO includes server purchasing cost plus electricity cost for powering and cooling the servers. We assume electricity cost to be $0.07/kWh and we assume a three-year depreciation cycle. For the cooling cost, we assume there is need for 1 Watt of cooling power for each Watt of consumed power. The three-year total cost of ownership (TCO) for 6 Web servers, 1 memcached server and 2 databaser servers consists of hardware cost, power cost and cooling cost; this makes $24,887 for the hardware vendor's suggestion, $21,201 for suggestion #1 and $17,428 for suggestion #2. This means a reduction in TCO of 14.8% and 30.0% for suggestions #1 and #2, respectively. The performance per TCO metric leads to a gain of 6.7% for suggestion #1 and a loss of 10.0% for suggestion #2.

- **Performance per TCO^2 :** If total cost of ownership is more important than performance, performance per TCO-squared might be an appropriate metric. Using this metric, it is clear that there is a big benefit in using our two suggestions compared to the hardware vendor's suggestion, with a respective gain of 25.2% and 28.5%.

Note that total (both static and dynamic) power consumption is likely to be reduced even more because of reduced operating temperature which reduces leakage power consumption. In other words, the reduced cost factors mentioned above are pessimistic cost savings; actual savings in power consumption and total cost of ownership will be higher.

In summary, this case study illustrated evaluating hardware design choices in the data center, enabling service providers, data center owners, as well as system architects to make trade-offs taking into account end-user performance of a Web 2.0 workload.

7.2 Software optimizations

Whereas the first use case considered a hardware design trade-off, our second use case illustrates the potential for driving software trade-offs and analyses. The reason why this is valuable is that setting up such experiments in a live data center is considered to be too risky because it might interrupt normal operation. Our approach on the other hand allows for setting up such experiments in a controlled environment while being able to apply real-life user requests.

In this case study, we analyze the performance for an alternative Web server software package. As mentioned before, Netlog uses the Apache Web server software to process all user requests on the Web servers. Another well-known web server software package is called NGINX[18]. In Figure 17, we show the percentage increase in the number of requests that were handled under 300 ms, the chosen metric for this case study. We compare different NGINX configurations to the standard Netlog Apache configuration. On the horizontal axis we distinguish several NGINX configurations, starting with the default configuration on the left side.

We observe that replacing Apache by a default NGINX setup increases the number of requests handled under 300 ms by 7.5%. This number gets up to 13.5% when tuning the number of connections per worker thread. We also disabled the HTTP keepalive feature[19], but conclude that there is no performance difference in disabling this feature.

NGINX reduces response times as perceived by the end user, thereby increasing customer satisfaction. This will lead to more users visiting the social network site, leading to an increase in company profit. System engineers and software developers can easily study other software tweaks or parameter tuning for maximizing performance in the data center by using the proposed method.

8. RELATED WORK

8.1 Data center workloads

A number of studies have been conducted recently to understand what hardware platform is best suited for a given data center workload. In all of these setups, a single server is considered — the study focuses on leaf-node performance — and/or microbenchmarks with specific behavior are employed. In contrast, this paper considers a setup involving multiple physical servers running real workloads, and we focus on end-user performance.

Kozyrakis et al. [6] consider three Microsoft online services, Hotmail, Cosmos (framework for distributed storage and analytics) and the Bing search engine, and their goal is to understand how online services and technology trends affect design decisions in the data center. They collect performance data from production servers subject to real user input, and in addition, they set up a slightly modified version of the software in a lab setup in order to perform stress tests for evaluating individual server performance under peak load. Our work differs from the Kozyrakis et al. work in two important ways: (i) we consider a different workload (Web 2.0 social networking), and (ii) our methodology is very different: our lab setup includes multiple servers, running unmodified production software supplied with real-life user input; in addition, we focus on end-user perceived performance.

[18]http://www.nginx.net/ – High performance Web server with low memory footprint.

[19]The 'keepalive' feature is used for actively maintaining a connection between a client and a server.

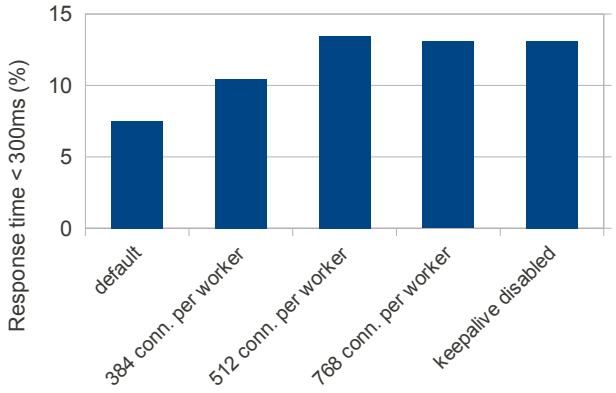

Figure 17: Increase in number of requests handled under 300 ms for different NGINX configurations, compared to the Apache Web server.

Lim et al. [7] consider four Internet-sector benchmarks, namely Web search (search a very large dataset within sub-seconds), webmail (interactive sessions of reading, composing and sending emails), YouTube (media servers servicing requests for video files), and mapreduce (series of map and reduce functions performed on key/value pairs in a distributed file system). These benchmarks are network-intensive (webmail), I/O-bound (YouTube) or exhibit mixed CPU and I/O activity (Web search and mapreduce). Lim et al. reach the conclusion that lower-end consumer platforms are more performance-cost efficient — leading to a 2× improvement relative to high-end servers. Low-end embedded servers have the potential to offer even more cost savings at the same performance.

Andersen et al. [1] propose the Fast Array of Wimpy Nodes (FAWN) data center architecture with low-power embedded servers coupled with flash memory for random read I/O-intensive workloads. Vasudevan et al. [10] evaluate under what workloads the FAWN architecture performs well while considering a broad set of microbenchmarks ranging from I/O-bound workloads to CPU- and memory-intensive benchmarks. They conclude that low-end nodes are more energy-efficient than high-end CPUs, except for problems that cannot be parallelized or whose working set cannot be split to fit in the cache or memory available to the smaller nodes — wimpy cores are too low-end for these workloads.

Reddi et al. [8] evaluate the Microsoft Bing Web search engine on Intel Xeon and Atom processors. They conclude that this Web search engine is more computationally demanding than traditional enterprise workloads such as file servers, mail servers, Web servers, etc. Hence, they conclude that embedded mobile-space processors are beneficial in terms of their power efficiency, however, these processors would benefit from better performance to achieve better service-level agreements and quality-of-service.

8.2 Sampling

Sampling is not a novel method in performance analysis. Some of the prior work mentioned above focuses on leaf-node performance, an example of sampling in space. Sampling in time is heavily used in architectural simulation. Current benchmarks execute hundreds of billions, if not trillions, of instructions, and detailed cycle-accurate simulation is too slow to efficiently simulate these workloads in a reasonable amount of time. This problem is further exacerbated given the surge of multi-core processor architectures, i.e., multiple cores and their interactions need to simulated, which is challenging given that most cycle-accurate simulators are single-threaded.

Sampled simulation takes a number of samples from the dynamic instruction stream and only simulates these samples in great detail. Conte et al. [4] were the first to use sampling for processor simulation. They select samples randomly and use statistics theory to build confidence bounds. Further, they quantify what fraction of the sampling error comes from the sampling itself (sampling bias) versus the fraction of the error due to imperfect state at the beginning of each sample (non-sampling bias or cold-start problem). Wunderlich et al. [11] employ periodic sampling and very small samples while keeping the cache and predictor structures 'warm', i.e., cache and predictor state is simulated, while fast-forwarding between samples.

Whereas both the Conte et al. as well as the Wunderlich et al. approaches select a large number of samples and rely on statistics to evaluate the representativeness of the samples, Sherwood et al. [9] employ knowledge about program structure and its execution to determine representative samples. They collect program statistics, e.g., basic block vectors (BBVs), during a profiling run, and they then rely on clustering to determine a set of representative samples. The approach taken in this paper is similar to Sherwood et al. although we take different workload statistics as input to the sample selection algorithm, while considering server workloads rather than CPU workloads.

9. CONCLUSION

In this paper, we presented a case study in which we characterized a real-life Web 2.0 workload and evaluated hardware and software design choices in the data center. Our methodology samples the Web 2.0 workload both in space and in time to obtain a reduced workload that can be replayed, driven by input data captured from a real data center. The reduced workload captures the important services (and their interactions) and allows for evaluating how hardware and software choices affect end-user experience (response times).

The real-life Web 2.0 workload used in this work is Netlog, a popular and commercially deployed social networking site with a large user base in Europe. We explored hardware trade-offs in terms of core count, clock frequency, HDD versus SSD, etc., for the Web, memcached and database servers, and we obtain several interesting insights, such as the Web servers scale well with core count, and end-user response times are inversely proportional to Web server CPU frequency; an SSD reduces the longest response times by around 30% over an HDD in the database servers, which may or may not be justifiable given the significantly higher cost for SSD versus; memcached servers show low levels of CPU utilization, and are both memory and network-bound, hence, hardware choice should be driven by the cost of integrating more main memory in the server. Further, we presented two case studies illustrating how the method can be used for guiding hardware purchasing decisions as well as software optimizations.

Acknowledgements

We thank the anonymous reviewers for their constructive and insightful feedback. Stijn Polfliet is supported through a doctoral fellowship by the Agency for Innovation by Science and Technology (IWT). Frederick Ryckbosch is supported through a doctoral fellowship by the Research Foundation–Flanders (FWO). Additional support is provided by the FWO projects G.0255.08, and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

10. REFERENCES

[1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the International ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, Oct. 2009.

[2] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar. 2003.

[3] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2009.

[4] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 468–477, Oct. 1996.

[5] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[6] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30:8–19, July/August 2010.

[7] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 315–326, June 2008.

[8] V. J. Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 26–36, June 2010.

[9] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, Oct. 2002.

[10] V. Vasudevan, D. Andersen, M. Kaminsky, L. Tan, J. Franklin, and I. Moraru. Energy-efficient cluster computing with FAWN: Workloads and implications. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking (e-Energy)*, pages 195–204, Apr. 2010.

[11] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, June 2003.