Workload Characterization of selected JEE-based Web 2.0 Applications

Priya Nagpurkar William Horn U. Gopalakrishnan Niteesh Dubey Joefon Jann Pratap Pattnaik
IBM T.J. Watson Research Center

Abstract

Web 2.0 represents the evolution of the web from a source of information to a platform. Network advances have permitted users to migrate from desktop applications to so-called Rich Internet Applications (RIAs) characterized by thin clients, which are browser-based and store their state on managed servers. Other Web 2.0 technologies have enabled users to more easily participate, collaborate, and share in web-based communities. With the emergence of wikis, blogs, and social networking, users are no longer only consumers, they become contributors to the collective knowledge accessible on the web. In another Web 2.0 development, content aggregation is moving from portal-based technologies to more sophisticated so-called mashups where aggregation capabilities are greatly expanded.

While Web 2.0 has generated a great deal of interest and discussion, there has not been much work on analyzing these emerging workloads. This paper presents a detailed characterization of several applications that exploit Web 2.0 technologies, running on an IBM Power5 system, with the goal of establishing, whether the server-side workloads generated by Web 2.0 applications are significantly different from traditional web workloads, and whether they present new challenges to underlying systems. In this paper, we present a detailed characterization of three Web 2.0 workloads, and a synthetic benchmark representing commercial workloads that do not exploit Web 2.0, for comparison.

1. Introduction

Web 2.0 represents the evolution of the web from a source of information to a platform [15]. This emerging platform has facilitated various participatory activities over the internet like social networking, collaboration, and content sharing. Enterprises are beginning to use Web 2.0 technologies to increase employee productivity by enabling similar collaborative activities in the corporate world. Web 2.0 applications use relatively new technologies to drive traditional Internet technologies in ways that are quite different from existing web and enterprise applications, presenting new challenges to underlying systems.

In spite of the excitement and interest that Web 2.0 has generated, to the best our knowledge, there has been no systematic characterization of these workloads and their impact on the underlying systems that host them. Such a characterization is an important first step towards driving innovation at the architecture, operating system, JVM, and middleware levels, to better support these emerging workloads. The lack of Web 2.0 benchmarks, or applications that are easy to deploy and run, is one of the major hindrances inhibiting more research in this area. In this paper, we first describe the methodology we used to set up existing Web 2.0 applications as benchmarks, and then present a detailed characterization of workloads generated using these benchmarks.

To create our benchmarks, we took two JEE-based[1] applications that exploit Web 2.0 features, and designed a test harness that exercises them in a manner representative of real-world use scenarios. We deployed these benchmark applications on an IBM Power5 system with the primary goal of analyzing the server-side behavior of these workloads, and finding optimization opportunities in the system and software stack that hosts them. We present a number of statistics for these benchmarks, from the user level behavior as captured by HTTP requests to micro-architectural behavior captured by hardware performance counters, and also contrast them with a traditional enterprise Java benchmark. Our analysis captures two expected features of Web 2.0 applications, namely their *chattiness* and the presence of *user-generated content* from the HTTP characteristics, and indicates that the *data-centric behavior* of Web 2.0 applications leads to a significant amount of time spent on data cache miss stalls.

In summary, we make the following contributions:

- Explanation of Web 2.0, especially from a Java server environment perspective
- A detailed description of how existing Web 2.0 applications can be turned into repeatable and realistic

[1] The Java Enterprise Edition platform (JEE) was formerly called J2EE, we use JEE in this paper to represent both

benchmarks

- A detailed characterization of three Web 2.0 workloads

The rest of the paper is organized as follows. We start with some background on Web 2.0 applications and technologies in Section 2. In Section 3 we describe the Web 2.0 applications we analyzed along with the test framework that we developed. We then present our analyzes of these workloads in Section 4, followed by related work in Section 5. We conclude in Section 6 with a summary of our findings and a discussion of future work.

2. Background

Since its inception, the Internet has permitted the exploitation of a variety of new applications. One of the applications, the World Wide Web, has permitted users to navigate an extensive network of interlinked resources and access large amounts of information in a variety of forms including text, images, and video. From the start, the thin client browser was a cornerstone of the World Wide Web. Over time, the power and sophistication of client browsers, backed by advances in underlying standards, has increased to the point where JavaScript-based client technologies permit a significant degree of interactivity. Also, advances in network technologies have permitted reliable connections to the World Wide Web at broadband speeds to become the norm.

Backed by these advances in the browser technologies and network performance and with the increasing sophistication of its users, the Web has also become increasingly more participatory. More and more users are contributing, sharing and collaborating on more and more content. Types of contributed content include: opinions that are uploaded and shared in blogs; collaborative authoring in wikis; and videos and images that are uploaded into and shared from searchable servers. There are many more examples.

Browser and network advances have also permitted the migration of client-side function from the Personal Computer (PC) to the server-side. Migrated function includes a large variety of applications from word-processing applications and email to specialized business applications offered using the Software-as-a-Service (SaaS) paradigm. Migrating function from the client PC to the server allows support of application health and user data to be aggressively managed at a central site. For example, an application on a central server can be deployed and thoroughly tested to a degree not feasible when the application is deployed on individual PCs. Similarly, the user data underlying an application can be stored and backed up with qualities of service not easily achieved on individual PCs. Also, applications and data can be accessed from any client connected to the web not just a particular PC.

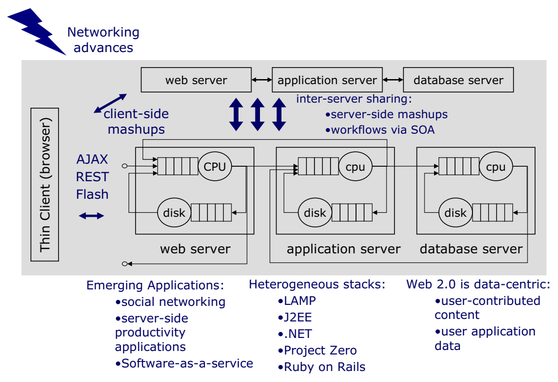


Figure 1: Web 2.0 Overview

There have also been many advances on the server-side. Commercial applications have developed interdependencies that permit sophisticated workflows to collaboratively deliver services to consumers and other businesses. Service Oriented Architecture (SOA) techniques permit interaction of loosely-coupled, distributed services across the internet. Loose coupling permits services that depend on different underlying environments (e.g., operating systems and middleware) to interoperate. Loose coupling also permits replacement of one service (e.g., a bulk mailer) with another, perhaps more attractive, service.

The increased sophistication of loosely-coupled, inter-server interactions is not limited to workflow. So-called server-side mashups permit servers to pull together information from a variety of sources to present a single view of the aggregated data to the user. For example, a server-side mashup might use the Internet to pull map data from one server, weather data from another server, and traffic data from yet another server to create a single, combined view for the user. Mashups can also be performed on the client-side, where the browser might present the user with content from multiple sources in a single window.

The term *Web 2.0* is commonly used to refer to the multiple advances in the World Wide Web covered in this section. In Figure 1, we've attempted to overlay a summary of these Web 2.0 features onto the traditional view of a multi-tier system composed of a web server, an application server, and a database server. On the left side of the figure we highlight the nature of the exchange between the thin client and web server. Technologies like AJAX and Representational state transfer (REST) are used to drive the Web 2.0 clients. FLASH is a popular technology for delivering video content. Along the top of the figure we depict other servers contributing to client-side mashups, server-side mashups, and distributed workflows. On the bottom left we highlight some of the emerging applications discussed in this section. In the middle bottom we point out that there are multiple stacks that are evolving to support Web 2.0. And finally at the bottom right we reinforce the point previously made that much of Web 2.0 is centered around server-side repositories of data.

Considering the trends outlined in this section we have identified three key areas where Web 2.0 will impact systems. First, the movement of function from the client-side to server-side has resulted in increased network interaction between the client and the server (i.e., “chattiness”). Second, to support SOA and server-side mashups, the frequency and nature of inter-server transactions are changing. Third, given the increasing degree of access to an increasing quantity of user-contributed data, the amount of server-side data is increasing and the nature in which it is accessed is changing.

In this paper, our primary goal is to analyze the impact of Web 2.0 workloads on the server-side. For benchmarks, we use a commercial application, Lotus Connections, and a JEE5 Web 2.0 benchmark Pet Store 2. The next section describes these applications and the framework we built to drive them.

3. Benchmarks and Workload Design

In this section, we describe the Web 2.0 applications that we used as our benchmarks, and the infrastructure used to drive them. We chose to characterize the Web 2.0 features of two Java-based applications: *Lotus Connections* and *PetStore 2.0*, which are described in the following subsections.

3.1 Lotus Connections

Lotus Connections is IBM’s JEE-based social collaboration software for the enterprise [9]. It consists of five component services:

- **Profiles**: The Profile component enables the user to use attributes such as name, organization, location, reporting structure, and interests to find other employees.
- **Communities**: The Communities component enables employees sharing a common interest to collaborate with one another.
- **Blogs**: The Blogs component provides a mechanism for employees to express their views and share their knowledge with other employees.
- **Dogear**: The Dogear component allows employees to organize and centrally store bookmarks on a server. Importantly, bookmarks can also be shared with other employees.
- **Activities**: The Activities component helps employees create and share tasks, to-dos, and best practices.

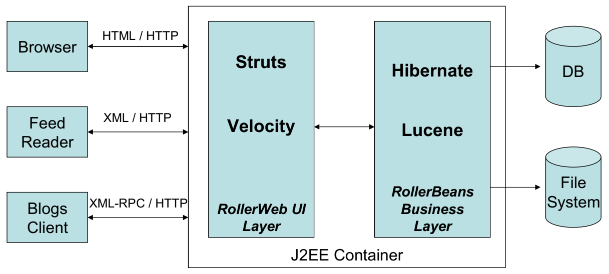


Figure 2: Lotus Connections Blogs Architecture

These five components can be used together or individually, and can be deployed on individual or shared instances of an application server. A relational database is used as the back-end data store. An LDAP server is used to provide user authentication and directory services. We chose to analyze the two components that rely most heavily on Web 2.0 features, these are Blogs and Dogears.

Blogs

Blogging has emerged as a powerful Web 2.0 tool for sharing information. Within an enterprise, blogs provide a mechanism for employees to share their knowledge and expertise. It provides a convenient, non-intrusive mechanism to present ideas and receive feedback from other interested parties. Lotus Connections uses a derivative of the Apache Roller blog engine. Apache Roller consists of two layers, a UI layer and a data layer. In the UI layer, the editor is implemented using Struts. Blog rendering is done using Velocity. In the data layer, Hibernate is used to provide relational persistance and Apache Lucene powers search. Atom feeds for blogs, blog entries and comments are also supported. Figure 2 shows the architecture of the Blogs component of Lotus Connections.

Dogear

Dogear is the social bookmarking component of Lotus Connections. It permits users to organize bookmarks with tags, to store them in a central server, and to share them with other users. Using Dogear, people can discover bookmarks that have been qualified by others with similar interests and activities. The Dogear component also supports Atom feeds.

3.2 Java Pet Store 2.0

Java Pet Store 2.0 [3] is a reference application for building AJAX-based Rich Internet Applications (RIA) on a JEE 5 platform. It is developed by the Java Blueprints program at Sun and source code is available for download. Java Pet Store 2.0 is an update of the well-known Java Pet Store application, and was developed to demonstrate JEE 5 support

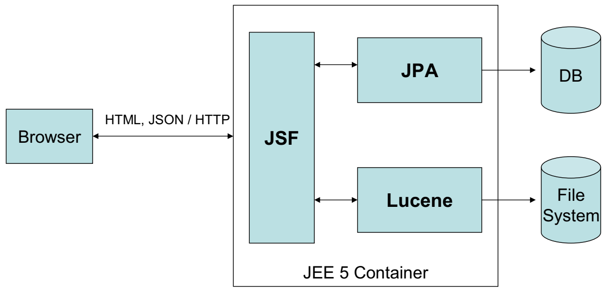


Figure 3: Petstore Architecture

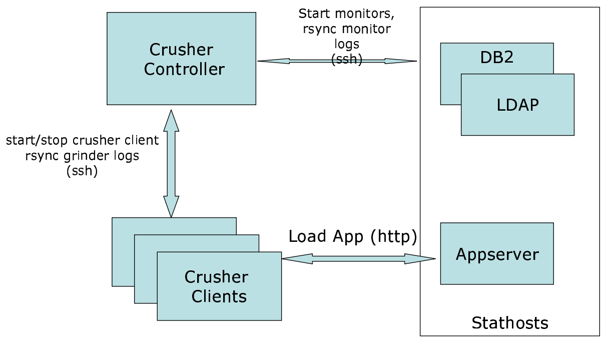

Crusher Controller
Start monitors, rsync monitor logs (ssh)
DB2
LDAP
start/stop crusher client rsync grinder logs (ssh)
Load App (http)
Crusher Clients
Appserver
Stathosts

Figure 4: Crusher Setup

for Web 2.0 features. Petstore 2.0 scenarios include listing pets for sale, searching for pets to buy, tagging pet listings, obtaining the geographic location of a pet for sale, and purchasing a pet using paypal. A feature for obtaining maps of geographic locations demonstrates a Web 2.0 mashup where services from multiple sources are combined to provide a new value-added service.

The implementation of Java Pet Store 2.0 follows the model - view - controller (MVC) design pattern and exploits technologies introduced in JEE 5, including the Java Persistence API (JPA) for model function and Java Server Faces (JSF) for view function. The view component demonstrates the use of AJAX in a JEE 5 environment to support a rich, interactive user interface. The AJAX components are implemented using the Dojo toolkit. Search is implemented using Apache Lucene. JavaScript Object Notation (JSON) is used for some of the data transfers between the server and the client browser. The architecture of the Petstore 2.0 application is shown in Figure 3.

3.3 Workload Harness and Data Collection Infrastructure

To facilitate driving our benchmarks, we developed an infrastructure, named Crusher, which extends the Grinder tool [2]. Grinder is an open-source load testing tool written in Java. It can use multiple Java threads, multiple JVMs, and multiple clients to generate load for an application. Grinder tests are easily written in the Jython scripting language and can drive any load that has a Java API (e.g., HTTP, SOAP, REST, RMI, JDBC).

Generating Testcases and Workloads

The Grinder includes a proxy-based tool for capturing browser-generated HTTP requests. We use this tool to capture raw HTTP requests generated by our applications. We then partition the raw HTTP requests into transactions. Each transaction represents a user-level action and can include multiple page requests. A workload consists of a *mix* of these transactions as specified by the user. The mixes we used for the data presented in this paper represent common usage patterns.

For the Lotus Connections workload, we used a deployment internal to IBM to derive these usage patterns. However, we are also interested in analyzing the effect of different workload mixes, and those that vary with time as part of future work. We obtained some of the content to populate the databases used by Lotus Connections also from the internal deployment. This content consists of 100 thousand users, 29 thousand of whom have blogs, and 16 thousand of whom have dogear bookmarks, with a total of 68 thousand blog entries and 168 thousand bookmarks. 65 thousand of the blog entries have comments. For PetStore 2.0, we used the sample data that comes with the application.

During a benchmark run, first a transaction is chosen randomly, based on the mix specification. Data needed for the transaction, like a username or a blog entry, is either randomly generated or randomly chosen from our dataset. To maintain repeatability, we undo the effects of write transactions (transactions that create new content), by restoring all databases to their initial state after every run. Table 1 lists the transactions for each of our applications along with their HTTP characteristics. Note that these characteristics vary slightly across different invocations of the same transaction, depending on the specifics of the request. For example, the number of bytes received as a result of a blogs *readEntry* transaction depends on the particular blog entry requested. The last column in Table 1 shows the percentage of that transaction in the mix we used to gather the data presented in Section 4.

Executing Workloads and Collecting Data

Crusher operates on *client nodes* and *log nodes*. To generate load, Crusher distributes the Grinder engine and workloads to the clients. Subsequently, for each client, Crusher starts the Grinder engine with a user-specified workload.

Crusher is controlled by a policy file called the *spray* file. The spray file specifies client nodes, various Grinder properties for each client (number of virtual users, sleep

Table 1: Transaction Characteristics. This table lists the transactions in our Web 2.0 workloads and their HTTP characteristics. For each application, a workload consists of a mix of its transactions. The last column lists the percentage occurrence of that transaction in the mix used to gather the data presented in the paper.

Transaction	Avg # GET Requests	Avg # POST Requests	Avg # AJAX Requests	Avg # Bytes Sent	Avg # Bytes Received	% in Mix
blogsHome	45	0	1	0.0	92595.5	27.6
viewEntry	14	0	1	0.0	33477.7	27.6
getFeed	1	0	0	0.0	3448.1	13.4
addComment	16	2	0	692.8	71744.7	10.7
search	7	0	0	0.0	32078.5	8.4
viewBlog	46	0	2	0.0	685351.1	5.5
yourBlog	25	0	1	62.8	17598.2	3.9
viewFeedEntry	14	0	0	0.0	32374.4	1.9
createEntry	24	1	6	2544.8	80348.7	0.9

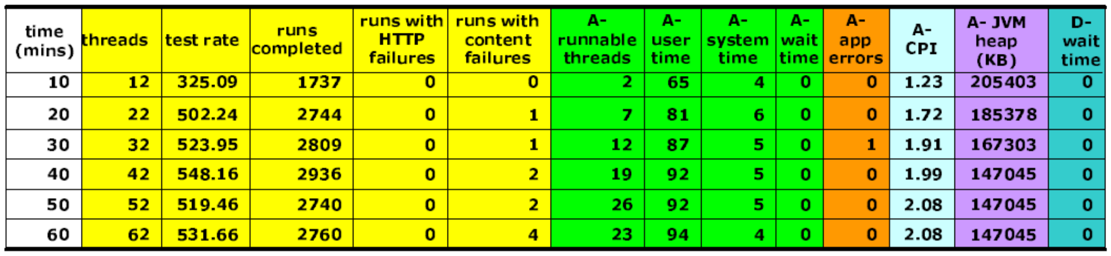
(a) Blogs

Transaction	Avg # GET Requests	Avg # POST Requests	Avg # AJAX Requests	Avg # Bytes Sent	Avg # Bytes Received	% in Mix
dogearHome	67	1	0	62.8	778503.6	19.1
dogearThis	20	0	0	0	6191.4	11.5
createBookMark	1	1	0	337	646.3	11.5
autoCompleteTag	0	0	1	0	291.5	9.0
myBookmarks	52	0	0	0	31076.4	7.6
popularTab	46	0	0	0	47295.5	7.5
getFeed	1	0	0	0	34173.6	6.5
nextPage	44	0	0	0	71422.2	6.2
autoCompleteMemberName	0	0	1	0	765.0	4.0
searchBookmarkPeople	48	0	0	0	61465.8	4.0
selectPersonTab	8	0	0	0	1895.4	4.0
searchBookmarkTags	46	0	0	0	51903.9	3.8
openTools	29	0	0	0	782861.6	3.7
getUserFeed	1	0	0	0	1840.0	1.7

(b) Dogear

Transaction	Avg # GET Requests	Avg # POST Requests	Avg # AJAX Requests	Avg # Bytes Sent	Avg # Bytes Received	% in Mix
home	29	0	0	0.0	642278.4	25.2
selectEntry	40	0	2	0.0	583983.6	20.4
catalog	37	0	2	0.0	558509.7	12.5
selectTag	24	0	0	0.0	244092.5	10.6
search	50	1	5	159.5	514142.0	9.9
map	6	0	0	0.0	176926.0	7.4
mapCategory	20	1	0	106.5	226475.0	7.4
tags	24	0	0	0.0	239322.3	5.3
seller	56	1	5	40317.3	643644.0	1.4

(c) PetStore



time (mins)	threads	test rate	runs completed	runs with HTTP failures	runs with content failures	A- runnable threads	A- user time	A- system time	A- wait time	A- app errors	A- CPI	A- JVM heap (KB)	D- wait time
10	12	325.09	1737	0	0	2	65	4	0	0	1.23	205403	0
20	22	502.24	2744	0	1	7	81	6	0	0	1.72	185378	0
30	32	523.95	2809	0	1	12	87	5	0	1	1.91	167303	0
40	42	548.16	2936	0	2	19	92	5	0	0	1.99	147045	0
50	52	519.46	2740	0	2	26	92	5	0	0	2.08	147045	0
60	62	531.66	2760	0	4	23	94	4	0	0	2.08	147045	0

Figure 5: Sample Report. The report is generated from multiple logs that track a variety of metrics on the machines in our distributed setup to provide a complete, one-glance view of a run.

time between requests, duration of the test run, number of processes, and number of threads), log nodes, and the type of statistics to be collected from each log host. Crusher uses ssh to drive the client and rsync to gather logs. Crusher is implemented in Jython and Python. A typical crusher setup is shown in Figure 4.

When requested by the user, Crusher collects statistics from the log nodes. The client nodes serve as log nodes for Grinder transaction logs. Crusher uses other log nodes to collect OS and middleware logs. Statistics collected and processed by Crusher include the basic client transaction statistics generated by Grinder, server-side CPU usage statistics from vmstat for both the application and database server (A- and D- in Figure 5), application server statistics from the Websphere Performance Monitoring Infrastructure (PMI) subsystem, server-side middleware error logs, and server-side logs containing hardware counters. After collecting logs from the log nodes, a Crusher tool is used to mediate the logs and output summary statistics into two spreadsheets. One of the two spreadsheets contains a variety of statistics for user-defined intervals. Figure 5 shows a sample output containing some of these statistics. The other spreadsheet contains summary statistics for each HTTP request used in the specified workload.

4. Analysis

In this section we present our analysis of the benchmarks described in Section 3. We begin with a description of our experimental setup and methodology, and follow with a detailed analysis at different levels in the system stack. In order to understand how our JEE-based Web 2.0 benchmarks relate to current enterprise server benchmarks, we also include results for Trade6 [5], a JEE online brokerage benchmark developed by IBM.

4.1 Experimental Methodology

The benchmarks analyzed in this paper run in a traditional three-tier configuration with a IBM DB2 9.1 back-end tier and a client/driver front-end tier. The middle tier runs the benchmark applications and we report results for this tier only. The benchmark applications are deployed on an application server. We use the IBM WebSphere Application Server ND 6.1.0.13 for our Lotus Connections benchmark and the GlassFish 9.1 application server [1] for the Pet Store 2.0 benchmark. Both application servers are run in the IBM Java Virtual Machine [10] with a heap size of 512 MB.

For the data presented in this paper, we run the middle tier on a 2-socket, 4-way Power5 multiprocessor running AIX 5.3, configured with 8GB of DRAM. The remaining components, including the database and LDAP servers, and the Grinder clients, are run on machines dedicated to generating load. Figure 6 illustrates our experimental setup and also shows details about the hardware and software used.

Given the multi-machine setup and three-tier configuration of our benchmarks, we had to spend considerable effort in tuning different components (e.g., application server thread pool sizes) in our system, to ensure that middleware and infrastructure bottlenecks do not distort our analysis, and that we are able to keep the system under test at least 95% busy. As such, our setup represents the state-of-the-art in enterprise application deployment.

We chose the number of virtual Grinder users required to load the system by observing the throughput of the workload for different numbers of virtual users. As seen in Figure 7, the throughput curve flattens out after an initial increase as more users are added. For each workload, we pick a point on the relatively flat portion of the curve, following the knee, to select the number of virtual users to use. The jagged nature of the PetStore curve is a result of concurrency issues that cause transaction roll-backs as the number of write requests increases. This problem needs further investigation and might require modifications to the application code. For the rest of the data in this paper, we used 40 virtual users for blogs and dogear, and 15 for PetStore.

4.2 CPU Utilization

Figure 8 shows the CPU utilization on the server machine when running the workloads with the mixes outlined in Section 3 and the number of virtual users determined above. It is necessary to ensure that the machine hosting

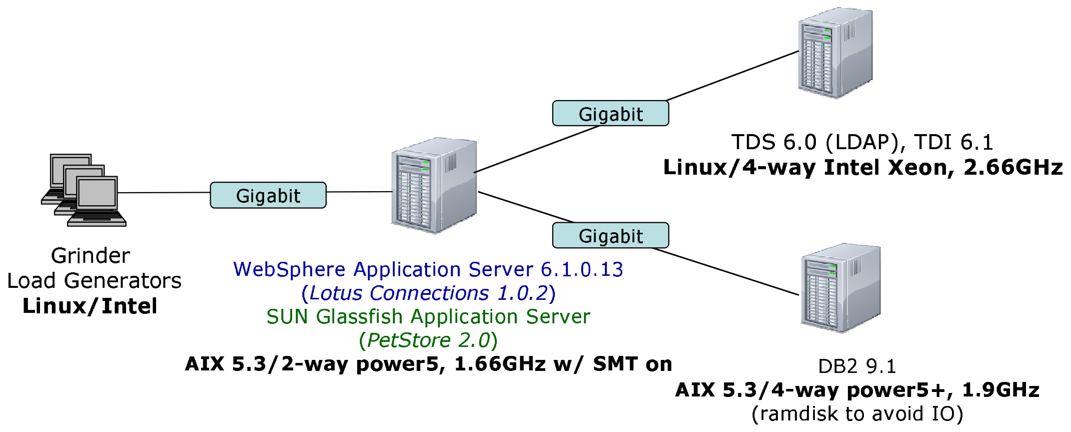

Gigabit
Gigabit
Gigabit
TDS 6.0 (LDAP), TDI 6.1
Linux/4-way Intel Xeon, 2.66GHz
Grinder
Load Generators
Linux/Intel
WebSphere Application Server 6.1.0.13
(Lotus Connections 1.0.2)
SUN Glassfish Application Server
(PetStore 2.0)
AIX 5.3/2-way power5, 1.66GHz w/ SMT on
DB2 9.1
AIX 5.3/4-way power5+, 1.9GHz
(ramdisk to avoid IO)

Figure 6: Experimental Setup.

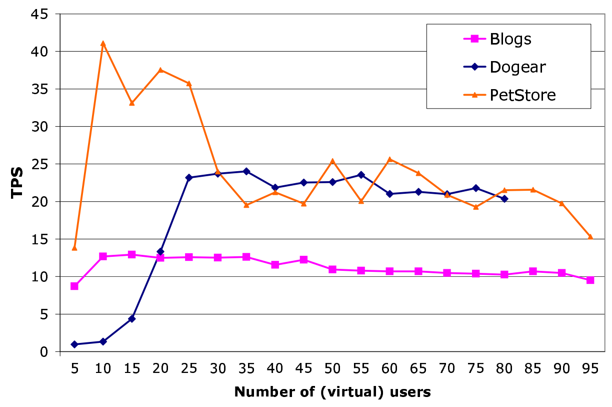


Figure 7: Throughput vs. Virtual Users. This Figure shows the transactions per second on the Y axis as the number of virtual users is gradually increased.

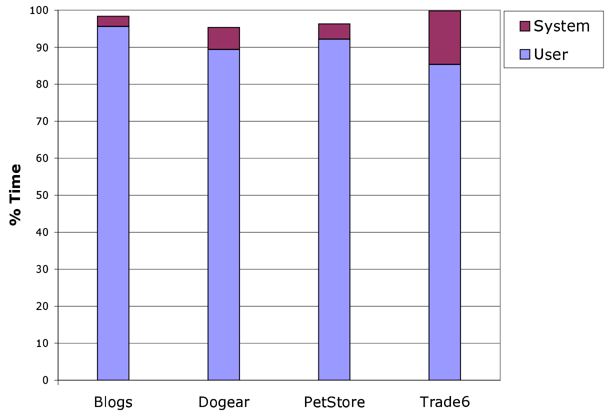


Figure 8: CPU Utilization. CPU utilization on the Application Server broken down into percentage time spent in user mode and system mode.

Table 2: Workload Characteristics. This table primarily lists the HTTP characteristics of our workloads. Rows 2 to 6 represent per transaction statistics. The bytes sent and received metrics capture the data flow from the client to the server.

	Blogs	Dogear	PetStore	Trade6
TPS	9.00	19.8	18.39	645
GET Requests/txn	22.38	30.27	31.15	1.14
POST Requests/txn	0.22	0.31	0.19	0.06
AJAX Requests/txn	0.70	0.13	0.89	0
Bytes Sent/txn	99.80	50.84	570.35	1.5
Bytes Received/txn	85172.73	195597.66	478720.54	10342
Code Size (MB)	20.7	18.8	35.5	26.4

the system under test is sufficiently busy so that the gathered data captures the behavior of the workload. This is especially true for hardware performance counter measurements. For all results we start with a 30 minute warm up period and follow with a 30 minute measurement period. As seen from the figure, the CPU is at most 5% idle and most of the time is spent executing user-level code. Compared to Trade6, the Web 2.0 workloads spend less time executing system-level code. We did not perform further analysis at the Operating system level since most of the time is spent in user-level code.

4.3 User and Program-level Characterization

Table 2 lists some HTTP and application level statistics collected over the 30 minute measurement interval. The first row lists transactions per second, followed by rows listing the total number of http GET, POST and AJAX requests per transaction, and the amount of data exchanged (excluding headers) with the server per transaction. The last two rows give an estimate of the memory footprint of these programs in terms of dynamic code size. Code size is calculated by tracking classes loaded by the JVM and includes code from Java libraries and the application server. All benchmarks, except PetStore 2.0, are deployed on the same system stack. The application server for PetStore is different from the

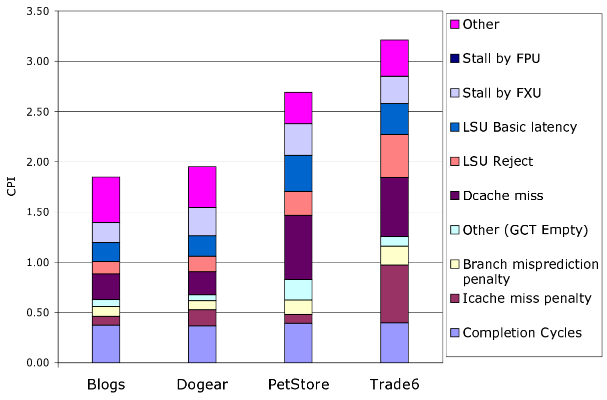


Figure 9: Stall Breakdown

other benchmarks and could lead to some differences in the use of runtime services and libraries. The *chattiness* of our Web 2.0 workloads is indicated by the presence of AJAX requests, and by the increase in number of HTTP GET and POST requests. There is also a marked increase in the amount of data flowing from the client to the server in the form of user generated content. PetStore makes the heaviest use of AJAX, and also has the largest amount of data exchange with the server. Note that AJAX requests also contribute to the traffic from the client to the server, although the size of each AJAX request is relatively small. Trade6 only has a single POST request used for user login and no other significant data flowing from the client to the server as seen in the table.

4.4 Stall Breakdown

As seen above, the user and application-level characteristics illustrate differences between the Web 2.0 workloads and the Trade6 workload. We used microarchitectural results to evaluate how these differences manifest themselves at the architectural level. These results are generated using the Power5 Performance Monitoring Unit (PMU). The PMU can be programmed to measure up to four concurrent processor events at a time, using dedicated per-thread counters. From the numerous events that can be tracked using these counters, we choose a subset of interesting ones. Performance counters are sampled after the 30 minute warmup period, with three, 30-second sampling intervals for each set of counters. We use the average counter value (across the three runs) to compute results.

The PMU counters can be programmed to be incremented at each stall of the processor's commit stage, such that each counter corresponds to the cause of the stall. Using these counters, we calculate the cycles per instruction (CPI) for each of our workloads, and break it down into its components. This allows us to understand how the cycles are being spent, and more interestingly, where the stalls are coming from. Figure 9 shows the CPI breakdown using a stacked column format. The height of the column shows the CPI. The bottom-most component represents useful work, i.e., cycles spent in successfully completing instructions, whereas the rest of the components represent processor stalls. For all workloads analyzed, data cache misses account for a significant portion of the stalls. For the Web 2.0 workloads, stalls due to instruction cache misses and branch mis-prediction are insignificant, whereas for the Trade6 workload, instruction cache misses account for almost as many stalls as data cache misses. The difference in instruction cache behavior is indicative of relatively smaller instruction working set sizes for the Web 2.0 workloads, compared to Trade6.

Table 3: Instruction Mix. This table shows the percentage of load, store, branch, and floating point instructions executed for the benchmarks under study.

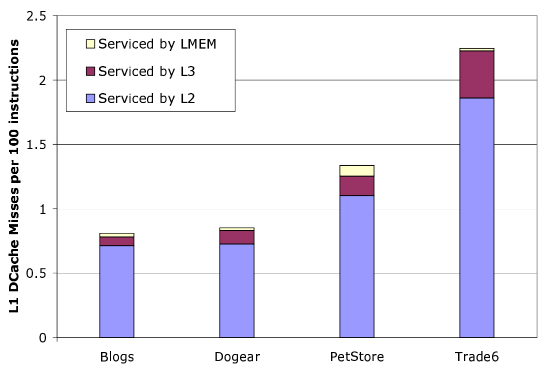
	Blogs	Dogear	PetStore	Trade6
% loads	25.91	22.96	30.78	29.42
% stores	12.18	13.15	16.54	18.74
% branches	24.64	22.58	21.80	20.09
% FPU	0.06	0.15	0.32	0.30

Table 3 lists the percentage of load, store, branch, and FPU instructions found in the instruction stream. We can see from the table, that the instruction mix for these applications does not vary significantly.

4.5 Cache Behavior

Figure 10 provides additional insight into the data and instruction cache miss components of the stalls by quantifying the miss rates and indicating where they are serviced from (access latency). For all workloads, most of the cache misses are serviced from the 1.8 MB shared L2 cache, with a smaller fraction being serviced from the off-chip 36 MB L3 cache. The Pet Store workload has a comparatively high number of data cache misses being serviced from the local memory, leading to a higher number of data cache miss stalls, in spite of lower miss rates, compared to the Trade6 workload.

To summarize, transactions in the Web 2.0 workloads analyzed differ significantly from those in traditional OLTP workloads like Trade6, in the number of HTTP requests and amount of data transfer that they generate. At the architectural level, stalls due to data cache misses form the dominant component of stall cycles. Data cache behavior characterization of PetStore (relatively high number of requests to memory) seems to indicate a need for better data prefetching algorithms. Contrary to behavior exhibited by other Java enterprise applications, like Trade6 and SPECjAppServer [14] instruction cache misses do not account for a significant portion of stall cycles.



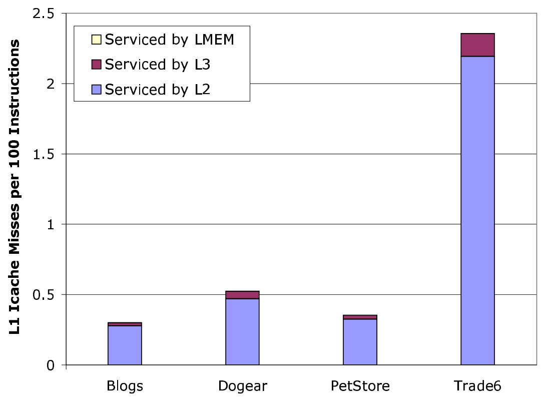


Figure 10: Cache misses and where they are serviced from. The graph on the left plots L1 Dcache misses and where they are serviced from, and the graph on the right similarly plots L1 Icache misses.

5. Related Work

We are not aware of any previous work on Web 2.0 workload characterization. Since the benchmarks we studied in this paper are JEE-based, the closest body of related work is the characterization of commercial Java workloads and application servers.

Several papers have focused on analyzing and optimizing enterprise Java benchmarks like Trade6, SPECjAppServer [7] and other Java OLTP workloads, especially at the architectural level [11, 14, 12, 18, 4]. Kunkel et al. described a general methodology for analyzing the performance of servers running commercial workloads [13]. Jann et al. compared the architecture and OS-level characteristics of Trade6 on two different IBM pSeries servers with the aim of finding bottlenecks and possible improvements in the OS [12]. Tseng et al. studied the scalability, with respect to cores and threads, of several commercial workloads running on Sun Microsystem's Niagara system [18]. Nagpurkar et al. used hardware performance counters to study the behavior of Trade6 and SPECjAppServer2004 and presented an instruction prefetching algorithm after observing that instruction cache misses are a significant problem for the applications they studied [14]. Shuf and Steiner also analyzed the performance of SPECjAppServer, and like this paper, used hardware metrics [17]. The above papers are the most comparable to our system, since they are also based on IBM's WebSphere Application Server, and indeed, their findings are consistent with ours. But unlike our paper, they do not focus on Web 2.0 behavior.

Xian et al. investigated how JEE application servers degrade under stress [19]. Most of the experiments focus on garbage collection behavior, which does not degrade gracefully under load. Our paper focuses on the normal case instead of the exceptional case, uses Web 2.0 instead of traditional workloads, and offers a broader perspective and a richer set of metrics.

Some papers characterize single-host Java workloads. Dieckmann and Hölzle performed one of the earliest workload characterizations for Java [8]. They focused on object characteristics like lifetime, size, type, etc.. Shuf et al. characterized the memory subsystem behavior of the SPECjvm98 benchmarks and of pBOB, a precursor to the SPECjbb2000 benchmark [16]. Their paper is similar to ours in that it presents a breakdown of cache miss behavior. Blackburn et al. developed the DaCapo benchmarks, a suite of Java programs for performance evaluation [6]. Like our paper, their paper describes new benchmarks along with a characterization using both hardware and software metrics. None of these three papers addresses Java application server behavior, let alone Web 2.0, which is the focus of our work.

6. Conclusions and Future Work

In conclusion, we have made a detailed characterization of several applications that exploit Web 2.0 technologies running on a IBM Power5 system. While we have found some interesting differences between traditional JEE applications and Web 2.0 JEE applications, the server-side impact of these applications is not significantly or consistently different from that of Trade6. More analysis is required to isolate the effects produced by Web 2.0 features.

For future work we plan to perform more analyzes on our current workloads, especially to analyze the effects of the data-centric nature of Web 2.0 applications. We also plan to examine additional Web 2.0 workloads. In particular, we plan to examine workloads that have a heavy exploitation of inter-server communications exhibited by server-side mashups and workflows operating in a SOA, and workloads where the mix of Web 2.0 operations varies with time. Finally, we plan to evaluate other platforms (including non-Power systems) to assess whether or not certain architectures are better suited for Web 2.0 workloads.

Acknowledgments

We thank Martin Hirzel, and the anonymous reviewers for providing useful comments on this paper.

References

[1] The Glassfish Open Source Application Server. `https://glassfish.dev.java.net//`.

[2] The Grinder Load Testing Framework. `http://sourceforge.net/projects/grinder`.

[3] Java PetStore 2.0 Reference Application. PetStore 2.0. `https://blueprints.dev.java.net/petstore/`.

[4] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. *Special Issue: Proceedings of the 25th annual international symposium on Computer Architecture (ISCA '98)*, 26(3):3–14, 1998.

[5] IBM Trade Performance Benchmark. Trade6. `https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6`.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffman, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Frampton Daniel, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[7] Standard Performance Evaluation Corporation. Specjappserver2004 benchmark. http://www.spec.org/jAppServer2004/, 2004.

[8] Sylvia Dieckmann and Urs Hölzle. A study of allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[9] IBM Social Networking Software for the Enterprise. Lotus Connections. `http://www-306.ibm.com/software/lotus/products/connections/`.

[10] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VEE)*, 2004.

[11] Morris Marden Harold W. Cain, Ravi Rajwar and Mikko H. Lipasti. An architectural evaluation of java tpc-w. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[12] Joefon Jann, R. Sarma Burugula, Niteesh Dubey, and Pratap Pattnaik. End-to-end performance of commercial applications in the face of changing hardware. Technical Report 24418, IBM Research, November 2007.

[13] S. R. Kunkel, R. J. Eickemeyer, M. H. Lipasti, T. J. Mullins, B. O'Krafka, H. Rosenberg, S. P. VanderWiel, P. L. Vitale, and L. D. Whitley. A performance methodology for commercial servers. *IBM Journal of Research and Development*, 44(6):851–872, 2000.

[14] Priya Nagpurkar, Harold W. Cain, Mauricio Serrano, Jong-Deok Choi, and Chandra Krintz. Call-chain software instruction prefetching in j2ee server applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[15] Tim O'Reilly. Web 2.0 Compact Definition: Trying Again. `http://radar.oreilly.com/archives/2006/12/web-20-compact-definition-tryi.html`.

[16] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2001.

[17] Yefim Shuf and Ian M. Steiner. Characterizing a complex J2EE workload: A comprehensive analysis and opportunities for optimizations. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[18] Jessica H. Tseng, Hao Yu, Shailabh Nagar, Niteesh Dubey, Hubertus Franke, Pratap Pattnaik, Hiroshi Inoue, and Toshio Nakatani. Performance studies of commercial workloads on a multi-core system. In *International Symposium on Workload Characterization (IISWC)*, 2007.

[19] Feng Xian, Witawas Srisa-an, and Hong Jiang. Investigating the throughput degradation behavior of Java application servers: A view from inside a virtual machine. In *Principles and Practice of Programming in Java (PPPJ)*, 2006.