Browser Workload Characterization for an Ajax-based Commercial Online Service

Shu Xu, Bo Huang, Junyong Ding and Jinquan Dai

Intel China Software Center

{samuel.xu, bo.huang, jonathan.ding, jason.dai}@intel.com

Abstract—**The transition to cloud computing and SaaS is a disruptive trend where users can conveniently access the services through browsers at any clients. In addition, with the prevalence of Web 2.0 and AJAX techniques, a browser-based client can have complex application logic and fancy user interface that are comparable to traditional desktop applications. This paper reports the study of workload construction and characterization for browser-based clients, using the Ajax-based web client of Zimbra (a commercial online messaging and collaboration suite). By comparing the various workload behaviors across different Zimbra server datasets, different browsers and different client platforms, it presents the characteristics of a real-life web application, which has significant differences from existing browser benchmarks in the literature. In addition, the platform-independent and browser-independent design of our workload makes it portable across various clients. Finally, this paper also provides valuable insights to the browser internals by analyzing the workload execution, the browser memory footprint and the breakdown of browser sub-modules.**

I. INTRODUCTION

The transition to cloud computing and SaaS is a disruptive trend where users can conveniently access the services through browsers at any clients. Unlike the install-and-run model of traditional software, most popular Rich Internet Applications (RIA) are delivered through internet and run within web browsers, the dominant user interface in internet era. With the prevalence of Ajax (Asynchronous JavaScript and XML) technique, the browser-based client can have complex application logic and fancy user interface that are comparable to traditional desktop applications. This model greatly facilitates the service deployment, maintenance and upgrade, e.g. the always-beta of GMail and Yahoo's Zimbra Collaboration Suite (ZCS).

Consequently, the web browsers have become the dominant cloud client and will play more and more important roles. Latest supporting evidence is the second browser war [2] among IE (Microsoft), Firefox (Mozilla), Chrome (Google), Safari (Apple), Opera, and etc.

To understand the typical characteristics of web 2.0 services, many workload characterization works have been performed on server side [3][4][6]. However, the workload characterization on client side, in particular browsers, is still at the emerging stage. Although some evaluations have been applied to modern browsers (e.g. Nielson compared browsers' start-up time, JavaScript, rendering and Ajax performance via separated methodologies [7]; Sam Allen compared browsers' memory consumption [8]; John Resig compared the performance of browsers' JavaScript engine and DOM processing [9]), it is obvious that additional efforts on commercial applications are needed and more detailed analysis through looking into the browser sub-modules is required.

In fact, a browser behaves quite differently between running a micro browser benchmark and running a real-life web application. For example, the execution of a JavaScript engine and JITted JavaScript code highly dominates the running of V8 benchmark suite [1]; on the other hand, it is not necessarily the case when running commercial workloads, as demonstrated by our browser workload characterization using Zimbra Collaboration Suits (ZCS) Web Client.

The contributions of this paper are as follows:

1. It proposes a suite of representative real-life commercial workload that could be used to understand browser behavior on different clients, which offers a way to conduct practical comparison among modern browsers.
2. It provides a detailed quantitative workload characterization of web browser using a web 2.0 messaging and collaboration suite -- ZCS Web Client, via comparing two modern browsers with various browser configurations, using different server datasets and different clients. Those compare provide valuable insights to the browser internals for both browser vendors and web based service vendors.
3. It shows different browser behaviors between running a commercial web collaboration application and running a micro benchmark.

The rest of this paper is organized as follows. Section II gives an overview of related software components of the workload we construct, followed by the details about the workload construction. Section III provides the workload profiling data and corresponding workload characterization. Section IV presents the conclusion and future work.

II. WORKLOAD CONSTRUCTION

A. Workload Components

Components of our workload are distributed in both client and server sides within a local network, which can be divided into three categories: ZCS Server, client browser and browser automation driver.

ZCS is a groupware product created by Zimbra Inc. and later acquired by Yahoo in September 2007. ZCS deployment has been boosted from 6 million paid mailboxes at Jan 2007 to 40 million paid mailboxes at March 2009 [10]. Its significant popularity is the major reason why we select it as our target workload. ZCS consists of ZCS Server and ZCS Web Client. The ZCS Web Client is a full-featured, browser-based collaboration suite that supports email and group calendars using an Ajax web interface, which enables rich internet experience to process email, document, task and calendar. Currently, two versions of ZCS server are available: an open-source version, and a commercially supported version ("Zimbra Network") with closed-source components. Our workload choose open-source version ZCS server 5.0 for performance analysis. There are three different types of ZCS Web Client: Ajax desktop portal, HTML desktop portal and Mobile portal. Since our workload is constructed on desktop/Netbook and Ajax portal is the most popular ZCS Web Client, we use Ajax desktop portal in our workload.

Since Microsoft launched Internet Explorer (IE) 4 embedded in Windows Operating System in Oct. 1997, IE has been the dominant web browser for more than 10 years. Things have changed a little bit now, as Mozilla's Firefox has occupied 20% market share since Nov. 2008 and Google released its open source browser Chrome at Sept. 2008 [2]. Alan Grosskurth abstracted the major components of a modern browser [11] as described in Figure 1.

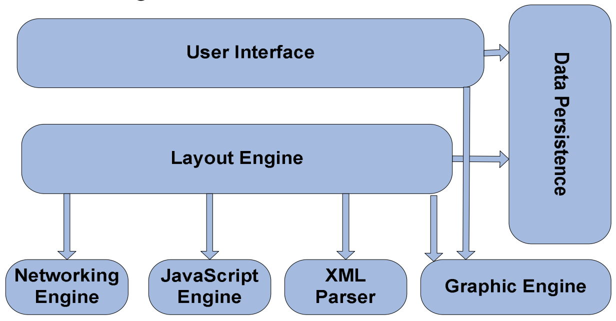

Figure 1 Abstracted modern browser architecture

Mozilla Firefox and Google Chrome are selected for study in this paper, since these mainstream open-source browsers bring a lot of freedom for us to understand the browser internals and conduct detailed performance analysis. The major components of Firefox 3.1b3 and Chrome 1.0154.43 are listed in Table 1. We also apply the same methodology to evaluate Internet Explorer 7.0.5730.13, and get execution time and peak working set data. However, since the detailed breakdown of Internet Explorer sub-modules is not available due to its close-source nature, this report does not contain the detailed characterization for Internet Explorer.

	Layout	JavaScript	Graphic	XML	Network
Firefox 3.1b3	Gecko	TraceMonkey/ SpiderMonkey	Cairo	Expat	Necko
Chrome 1.0154.43	Webkit	V8	Skia	LibXML	Internal "net" module

Table 1 Firefox and Chrome Module list

In order to minimize the performance variance caused by user behaviors, a browser-independent automation methodology is needed to emulate a client user by issuing the actions sequentially. Besides, a local ZCS server to minimize network latency is another key factor to minimize the performance variance. Basically, the automation could be done by either mimicking the GUI operations (e.g. keyboard or mouse events) at OS level with traditional tools such as VisualTest, or mimicking the user interactions (e.g. clicking a DOM element) at browser level with JavaScript. The latter is preferred because it has better understanding of the internal DOM structure of the browser and consequently makes it easier for us to determine the best timing to issue user actions. As a result, Windmill [12], an open-source automation web testing framework, is adopted to automate the interaction with browsers. A full Windmill architecture can be found at [17]. The basic idea of this utility is to use a broker proxy server to inject serialized user actions as JavaScript command to client browsers, without violating the same origin policy. All normal HTTP requests and responses between the browser and the web server traverse trough this proxy transparently. Most user actions in our workload are interleaved with DOM element synchronization, so as to ensure each action is a reasonable one against a valid DOM tree.

Figure 2 illustrates the workload topology and automation steps. For instance, if we want to automate a very simple Ajax scenario on a target web page (clicking button A to trigger the refreshing of text area B in the page and then waiting for the completion of the refreshing), the following sequence of events will happen between client browser and Windmill proxy server.

1) The client browser visits the web server through the proxy server, requesting the target web page.
2) The web server returns the HTTP response to the browser through the proxy server, which injects a JavaScript robot (JS-robot) into response and send it back to the client browser. From now on, the proxy server will not modify any HTTP request/response between the client browser and the web server.
3) The JS-robot in the response then has the client browser to display the target web page in the existing window, and to open a new window containing the JS-robot which drives the testing.
4) The JS-robot in the client browser then asks the proxy server whether there is any action for execution; and the proxy server sends "click button A" action (from its prepared script) back to the JS-robot for execution.

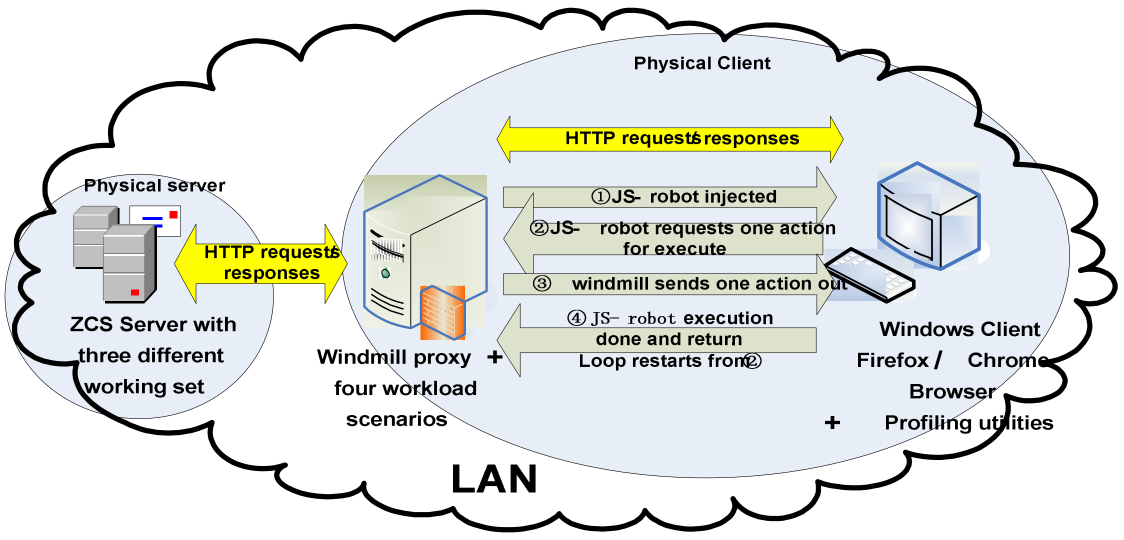
Physical Client
HTTP requests/responses
Physical server
HTTP requests/responses
①JS- robot injected
②JS- robot requests one action for execute
③ windmill sends one action out
④JS- robot execution done and return Loop restarts from②
ZCS Server with three different working set
Windmill proxy + four workload scenarios
Windows Client Firefox / Chrome Browser
+ Profiling utilities
LAN

Figure 2 workload topology and automation steps

5) The JS-robot issues a sequence of JavaScript code to mimic "click button A" on the target web page, then returns and asks the proxy server what the next action is to execute.
6) The proxy server sends the JS-robot an action of "wait for text area B refreshing" from prepared script.
7) The JS-robot periodically executes DOM exploration JavaScript code to check whether text area B becomes expected new content. Once new content is ready, it returns and asks proxy server what's the next action to execute.

Furthermore we have measured some user action sequences using both manual clicking and JS-robot triggering, and find those automation JavaScript only bring 0.8% and 1.15% additional JavaScript overhead (in terms of CPU cycles) in Chrome and Firefox respectively. As a result, the noise brought by our browser automation driver can be ignored in our experiments.

B. Workload Configuration

ZCS Open Source Edition v5.0 is installed on top of a 32-bit Redhat Linux EL4U6 (Dual Socket Intel Xeon 5130 dual core @ 2.0GHz, 1G Memory). We prepared 3 datasets each of which contains N emails, N address book contacts, N task items, N calendar items and N documents, where N is defined as 100/500/2000 for the small, medium and large datasets respectively. We collected ZCS server's CPU utilization data during workload running. Its average CPU utilization is less than 5.2% and peak CPU utilization is less than 39%. So ZCS server is not the bottleneck of whole workload environment. Since this study focuses on analyzing the client browser behavior, the server behavior and its interaction with client are out of the scope of this paper.

Since quad-core processor is predicted to be mainstream in desktop in 2009[18], our main client machine is an Intel Core Q9550 (quad cores @ 2.83 GHz, 3 G memory), with 32-bit Windows XP SP2 installed. Since Netbook has nice adoption trend, we also use ASUS 1000H (1.6GHz ATOM N270, 1G memory with 32-bit Windows XP SP2 installed) to verify the portability of our workload.

Firefox 3.1b3 browser is built from unmodified source code [13], under "MozillaBuild" environment [14], using Microsoft Visual Studio 2008 in-boxed compiler (version 15.0.21022.8) and linker (version 9.0.21022.8). The build configuration file removes most debug and test related flags except the symbol file generation flags. Since Firefox 3.1b3's JavaScript Just-in-Time (JIT) engine -- Tracemonkey can be turned on and off (i.e., switching to traditional JavaScript interpreter) through configuration change, we ran the workload in both modes.

Chrome 1.0154.43 browser is built from unmodified chromium source code [15] in Microsoft Visual Studio 2008 IDE environment, with the same version compiler and linker mentioned in Firefox building. We do not change the Chrome build flags which are contained inside Visual Studio project files. Normally Chrome has a main browser process and per-tab render process [16]. Chrome also supports running in single process mode to hold both main browser thread and render thread in ONE process. In our experiment, we run the workload in both modes: normal multi-process mode and single process mode.

Windmill proxy server is based on revision 1044 plus Chrome browser extension which contains Chrome support. The proxy server is installed at the same client machine (Figure 2), so that it can automatically launch and kill the browser.

We have defined four typical scenarios as follows.

1) Email: send/receive/delete/restore/tag/untag/search emails, navigate mail-boxes,
2) Task: search/navigate/open/close/tag/untag tasks
3) Address book: search/edit/navigate address book
4) Document: search/open/edit documents

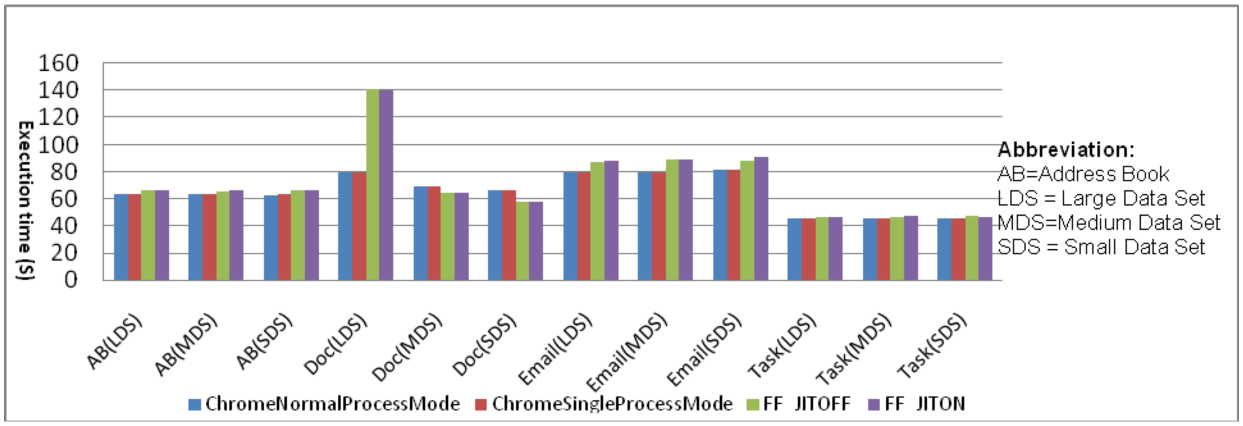

Figure 3 Execution time of Zimbra workload on desktop with different datasets (Lower is faster)

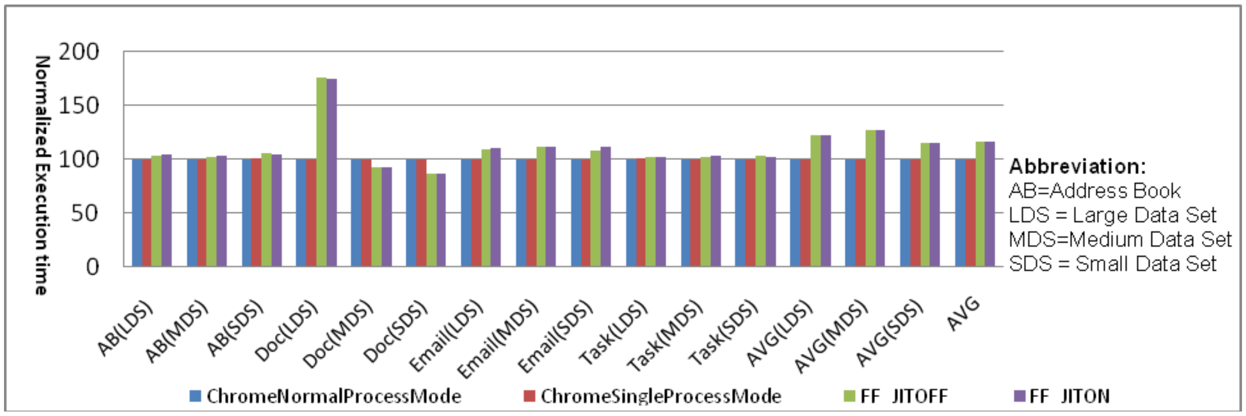

Figure 4 Normalized Execution time of Zimbra workload on desktop with different datasets (Lower is faster)

For each scenario, we first record its script using Windmill recorder, and then manually add the DOM element synchronization points in its script. The script for each scenario could be replayed using both Firefox and Chrome browser, with any of our (small/medium/large) ZCS dataset, without modifications. Each test scenario manipulates server data in a closure to eliminate the side-effect to subsequent test scenarios; e.g. if mails are sent or deleted in a scenario, the scenario will eventually undo those changes and restore the initial dataset. A new browser profile with empty cache content will also be generated before each round of scenario replay.

Because the calendar panel varies with the setting of machine date and time, it is not automation friendly. As a result, we have not designed the corresponding calendar scenario at the current stage.

C. Metrics

The following three types of profiling information are collected at client side for each test scenario:

1) **User action execution time:** The collection methodology is to record the start time and end time of a scenario execution, and derive the total duration of this scenario as the final execution time. Each scenario will be executed 6 times to get the average result. Both user login and logout time are excluded from the accumulated execution time. To better understand the impact to browser performance brought by the threading model, we also use Intel Thread Profiler (Version 3.1 build 26511) to dump the thread synchronization statistics.
2) **Memory footprint**: The collection methodology is to use a self-developed utility[1] to monitor the working set of a browser process at one second interval, so as to get the final working set curve and peak working set. For Chrome browser in normal multi-process model, both browser process and render process are monitored.
3) **Browser sub-module execution cycles and cycle breakdown**: The collection methodology is to profile each scenario using Intel VTune Performance Analyzer (Version 9.1 build 138) to collect CPU_CLK_UNHALTED.CORE event, and then map every event sample to each browser's sub-modules (with compiled symbol files). For Chrome browser, the CPU unhalt cycle breakdown information is only collected in single process mode.

[1] This utility is written by C. It firstly identifies expected browser process and then periodically invokes related Windows API to get the information of related process.

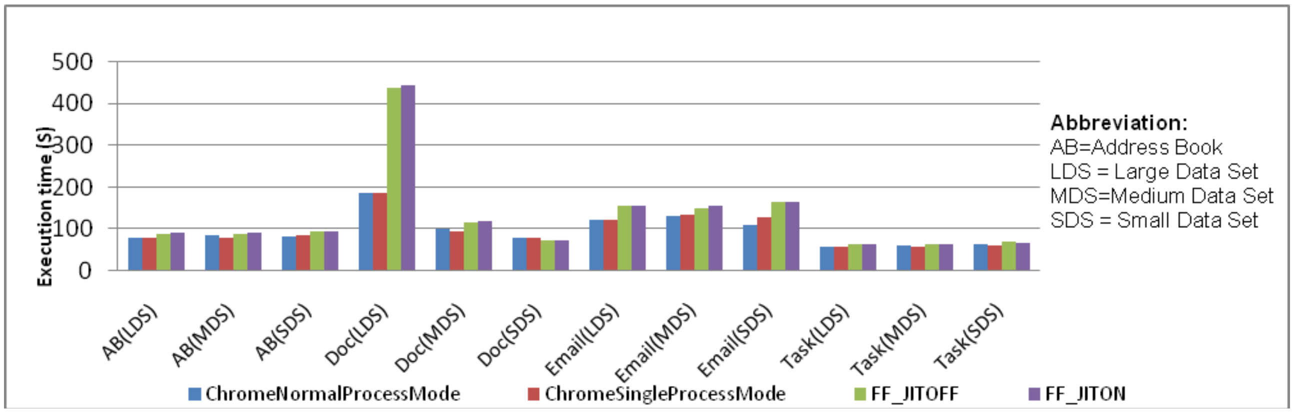

Figure 5 Execution time of Zimbra workload on Netbook with different datasets (Lower is faster)

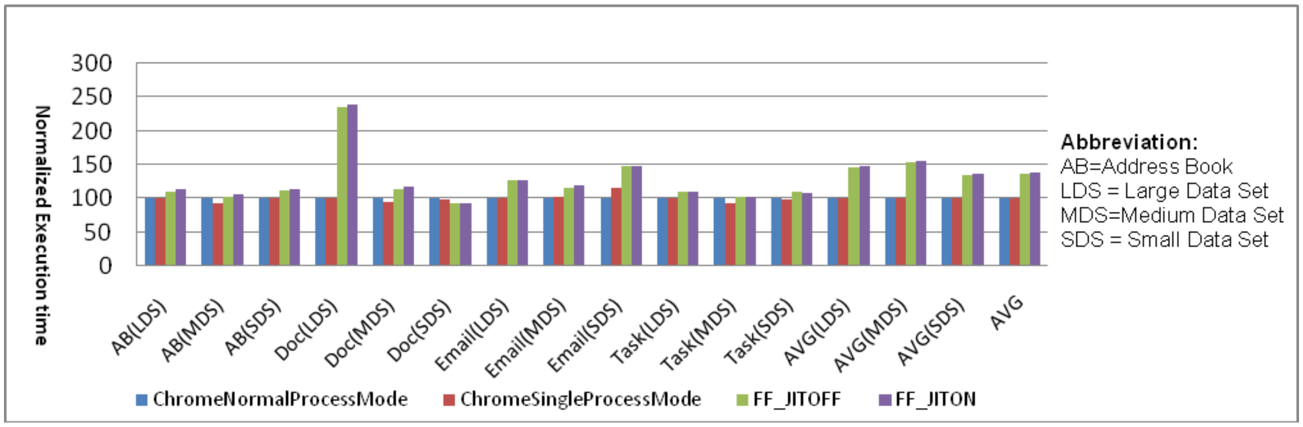

Figure 6 Normalized Execution time of Zimbra workload on Netbook with different datasets (Lower is faster)

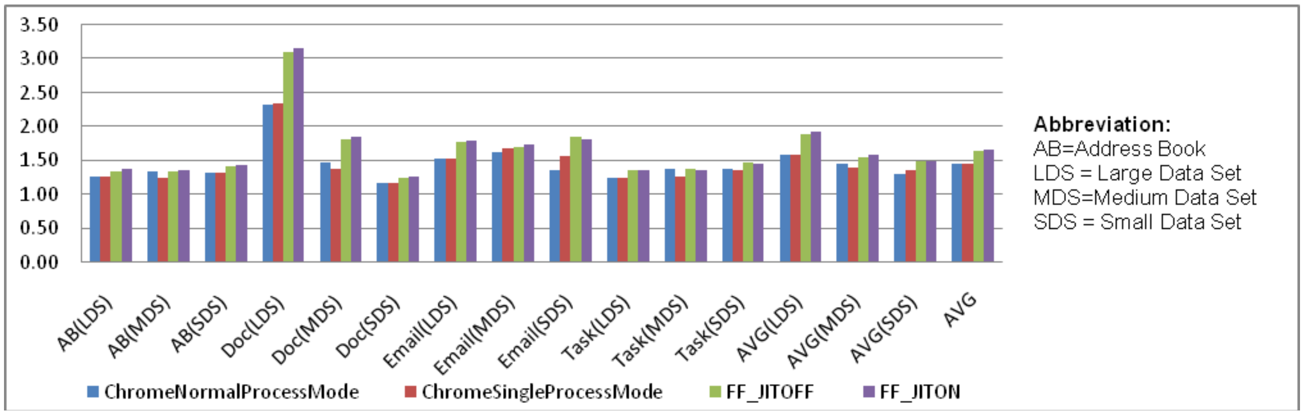

Figure 7 Desktop and Netbook's execution time slowdown ratio comparing

III. Experimental Result and Workload Characterization

A. Execution time result and characterization

Figure 3 and Figure 5 presents the execution time of ZCS web client workload (using small, medium and large ZCS server datasets) on Q9550 desktop and ASUS 1000H Netbook respectively. We can also find the normalized execution time from Figure 4 and Figure 6, which normalizes Chrome browser's execution time (in normal multi-process mode) to 100 for each scenario with a specific dataset.

From the data on execution time, we can characterize those 4 scenarios into 2 categories:

- **Dataset-sensitive scenario**: the scenario that shows big difference in execution time for different datasets. Documents scenario is an example.
- **Dataset-insensitive scenario**: the scenario that shows relatively flat execution time for different datasets. Address book, email and task scenarios are such examples.

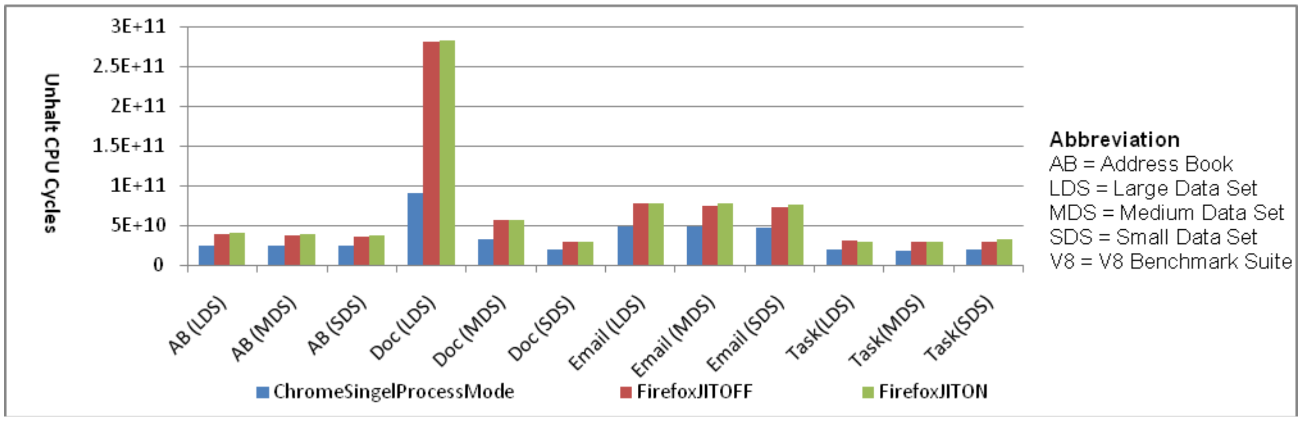

Figure 8 CPU unhalt cycle on desktop using different datasets

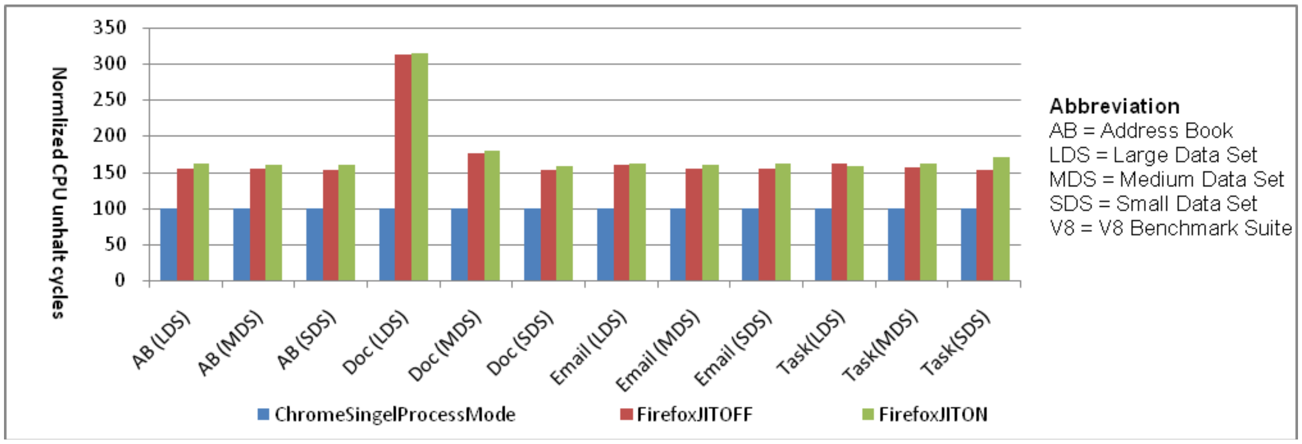

Figure 9 Normalized CPU unhalt cycle on desktop using different datasets

The major reason of the different behavior of document scenario on different datasets is that ZCS does not fold[2] the documents list, while DOES fold email, task and address book into specified number of items per page. Presenting a large document list consumes significant execution time in both browsers' layout module (Refer each browser's CPU cycle breakdown from Figure 10 - Figure 12).

We can also figure out that TraceMonkey doesn't bring obvious performance gain to Firefox, and sometimes it even leads to slight performance degradation. For instance, in Figure 3 "TraceMonkey OFF" has slightly shorter execution time than "TraceMonkey ON" on Email scenario with small dataset.

The reason for the negative performance impacts of TraceMonkey in those cases is that TraceMonkey's JITted code generation is based on the trace profiling in interpretation phase [5]. When TraceMonkey is turned ON, trace profiling overhead should be taken into consideration. In ZCS test scenarios, there are very tiny portion of JITted code got execution in most scenarios (refer to Figure 12 for CPU cycle breakdown of Firefox with TraceMonkey ON). The benefit from JITted code execution is less than the profiling overhead for the ZCS web client workload, which leads to the slight performance degradation.

[2] ZCS organizes all documents items into a single web page no matter how many documents there are. In contrast, ZCS organizes (folds) email/task/address book items in pages.

Comparing the performance between Firefox and Chrome, we can find that Firefox wins document scenarios with small and medium datasets no matter whether Tracemonkey is turned ON or OFF on desktop. On Netbook platform, Firefox only wins document scenarios with small dataset. Chrome wins all the other scenarios on desktop and Netbook. On average Chrome is faster than Firefox by 14.5% on desktop and 27% on Netbook from Figure 4 and Figure 6

There is a big performance gap of documents scenario with large dataset between Firefox and Chrome. The major reason of this performance gap can be found from CPU cycle breakdown of Firefox in Figure 11 and Figure 12, which show the layout engine execution of Firefox is significantly slower than Chrome's layout engine (as shown in Figure 10) to present large document list.

Figure 8 shows CPU unhalt cycles (CPU_CLK_UNHALTED.CORE event) on desktop with different datasets of Firefox and Chrome (in single process mode). Figure 9 normalizes those cycle data by normalizing Chrome browser's cycle number to 100 for each scenario with a specific dataset. We can find that the CPU unhalt cycles consumed by Chrome are average 2/3 of those consumed by Firefox on desktop; on the other hand, it does not bring corresponding 33% performance advantage to Chrome, and

Firefox even wins 2 scenarios on desktop. The major reason of that comes from Chrome's unique processing/threading mode. To reduce the response time of user interface for better user experience, Chrome asynchronously processes all blocking IO (e.g. network and disk) and other expensive operations using worker threads. Consequently, lots of cross-thread communications are needed [19][20]. Furthermore, there are also a lot of synchronizations between the main browser thread and render threads in Chrome [16]. According to the results obtained using Intel Thread Profiler, on average the number of Chrome's thread synchronization point in critical path is ~1.3x of Firefox. The heavy inter-thread synchronizations and communications have negative performance impact for Chrome browser.

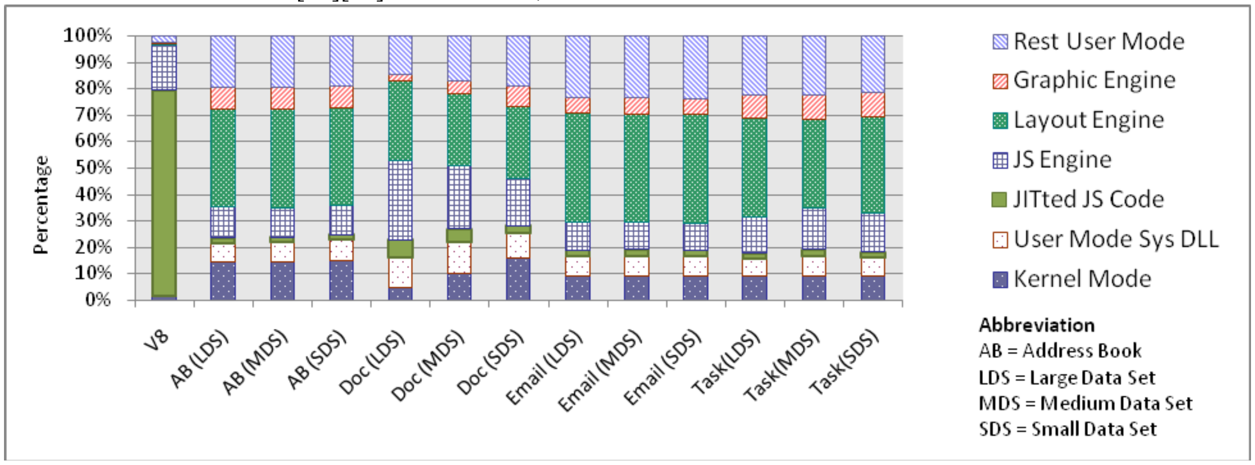

Figure 10 Module breakdown of Google Chrome Browser on desktop (Single process mode)

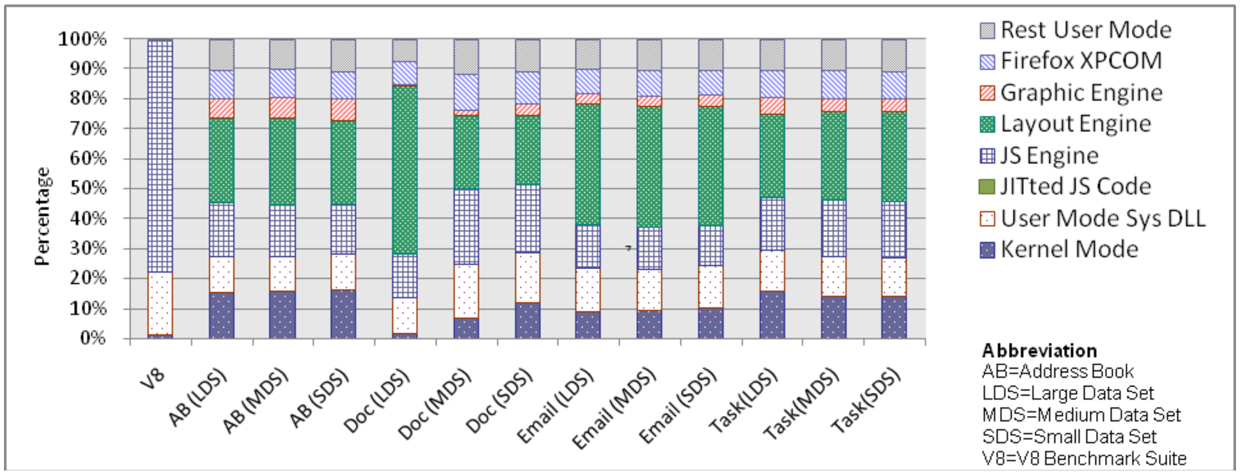

Figure 11 Module breakdown of Firefox on desktop (Tracemonkey OFF)

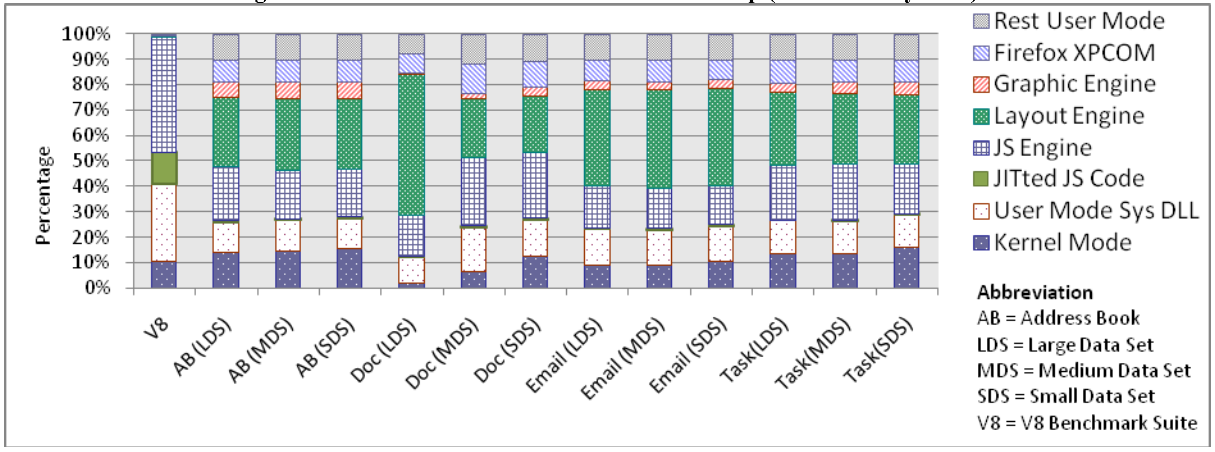

Figure 12 Module breakdown of Firefox on desktop (Tracemonkey ON)

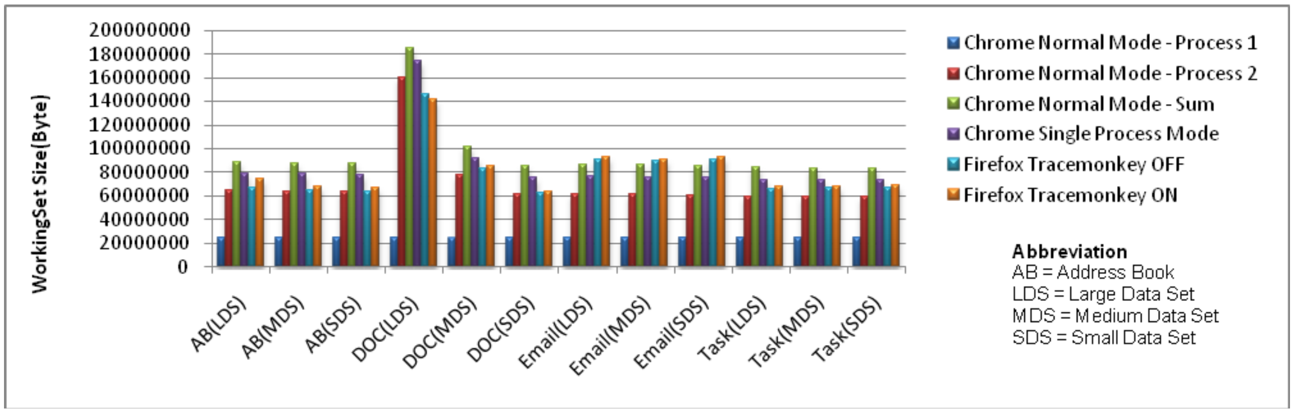

Figure 13 Peak memory working set compare on desktop

Figure 7 is derived from Figure 3 and Figure 5 to compare the desktop vs. Netbook slowdown ratio between Firefox and Chrome. It shows that Firefox suffers more severe slowdown than Chrome. The average Firefox slowdown ratio is 1.66:1 (Netbook execution time vs. desktop execution time), and average Chrome slowdown ratio is 1.45:1. In particular, for the document scenario with medium dataset, Firefox is faster than Chrome on desktop, while slower than Chrome on Netbook.

The major reason of different slowdown ratio is that Firefox is relative more CPU-intensive, and Chrome is relative more synchronization-intensive. When both of them migrated to a slower Netbook platform in our experiments, the execution time of the critical path of the more CPU-intensive Firefox is prolonged more severely.

B. Browser sub-module execution cycle breakdown and characterization

Figure 10 - Figure 12 show the breakdown of browser sub-modules, using the CPU_CLK_UNHALTED.CORE event sampling on desktop, for both V8 Benchmark Suite and ZCS web client workload (with different ZCS server datasets).

Obviously layout engine and JavaScript related modules (including JavaScript engine and JITted code) are the top 2 user mode sub-modules for both Firefox and Chrome browsers. Table 2 summarizes the distribution of those top 2 modules in Chrome and Firefox in more details. We can find that dataset-insensitive ZCS scenarios have stable distribution with different datasets, while dataset-sensitive ZCS scenario (document) has big variance with different datasets.

For V8 Benchmark Suite, JavaScript engine and JITted JavaScript code consume totally 58%-77% in Firefox's execution cycle and 95% in Chrome's execution cycle. While for ZCS web client JavaScript related modules only consume 13%-26% in Firefox's execution cycle and 13%-37% in Chrome's execution cycle. It is clearly that ZCS web client workload (a real-world, commercial web application) have very different behaviors from the V8 Benchmark Suite (micro benchmark).

1) In V8 Benchmark, JavaScript related module is the only performance dominator. Layout module normally consumes less than 1% CPU cycles.
2) In ZCS web client workload, the layout module is always the dominant module in Chrome browser on every scenario, which is also true for Firefox browser, except that JavaScript module dominates in document scenarios with small and medium datasets.

The JITted JavaScript code for Firefox has a tiny (normally less than 1%) ratio when TraceMonkey is turned ON for ZCS web client, as shown in Figure 12. This is because that TraceMonkey will consider a JavaScript loop as the trace JIT candidate only if it is executed more than once [5]. Due to the absence of loop inside ZCS web client JavaScript code, TraceMonkey does not generate a lot of JITted codes and most JavaScript code are interpreted. We can also observe that Chrome has a higher percentage of JITted code than Firefox. It is caused by Chrome's V8 JavaScript engine's different JIT policy: V8 compiles JavaScript source code directly into machine code when it is first executed [21].

This is an interesting finding that should be noticed by web application developers. That is, user experience of the ZCS web-based application depends more on the performance of the layout engine than the JavaScript performance in most cases. And rendering a very large HTML document consumes a lot of CPU cycle by the layout module, and can easily lead to poor user experience.

Chrome	**V8**	**AB**	**Doc**	**Email**	**Task**
Layout	<1%	37%	27%-30%	41%	35%±%2
JavaScript (JS engine +JITted code)	95%	13.5%±0.5	31%-37%	12%±0.5%	17%±2%
Firefox	**V8**	**AB**	**Doc**	**Email**	**Task**
Layout	<1%	28%±%1	22%-56%	39%±1%	29%±1%
JavaScript (JS engine +JITted code)	58%-77%	19%±%1	15%-26%	15%±2%	20%±2%

Table 2 Layout/JavaScript modules' cycle distribution on desktop

C. Memory footprint and characterization

Figure 13 compares peak memory working set of two browsers on desktop, using different execution modes and datasets.

It shows that the typical peak working set of ZCS web client is 60-100M bytes, while in some special cases (e.g. document scenario of large data set), the peak working set can reach 140M-185M bytes. In addition, the dataset-insensitive scenarios' peak working set are pretty stable, while a dataset-sensitive scenario (document scenario) has some variance with different datasets.

No matter in normal process mode or single process mode, Chrome has bigger memory footprint than Firefox for the scenarios of address book, document and task, and has less memory footprint for email scenario. Tracemonkey ON brings 1%-10% memory footprint increase in most cases except that turning on Tracemonkey reduces 2.5% working set in document scenario with large dataset.

IV. Conclusion and Future Work

This paper describes the mechanisms to construct a suite of representative real-world commercial workload that could be used to understand browser behavior, using the Ajax-based web client of Zimbra (a commercial online messaging and collaboration suite). The platform-independent and browser-independent design of our workload makes it portable across various clients.

It also presents the characteristics of the behaviors of Firefox and Google Chrome browsers with different ZCS server datasets, by comparing their execution time, CPU cycle breakdown and memory footprint on both desktop and Netbook. Based on the different behaviors of the browsers, we have analyzed how different threading model and CPU utilization in those two browsers impact their performance on desktop vs. Netbook, and have categorized the workloads into dataset-sensitive and dataset-insensitive scenarios.

In addition, this paper provides the detailed breakdown of the sub-modules of those two browsers, which shows that our ZCS web client workload (a real-world, commercial web application) has very different behaviors from the V8 Benchmark Suite (a micro benchmark). In particular, user experience of the ZCS web-based application depends more on the performance of the layout engine than the JavaScript performance in most cases. It is an interesting finding that should be noticed by web designers; that is, rendering a very large HTML document consumes a lot of CPU cycle by the layout module in a browser, and can easily lead to poor user experience.

Our future work will cover more usage scenarios using additional real-life web-based applications. We also plan to extend profiling metrics to micro-architecture, more OS related metrics and conduct more detailed internal analysis for browsers. In addition, we plan to work on workload construction and characterization for mobile browsers, for other client containers such as Adobe AIR. On the other hand, using those workloads to study the behavior of server(s) and the interactions between client(s) and server(s) would also lead to interesting researches.

References

[1] V8 Benchmark Suite - version 1 http://v8.googlecode.com/svn/data/benchmarks/v1/run.html.

[2] The second browser war http://en.wikipedia.org/wiki/Browser_wars#The_second_browser_war

[3] Nagpurkar Priya, etc. Workload characterization of selected JEE-based Web 2.0 applications. *In 2008 IEEE International Symposium on Workload Characterization*.

[4] Christopher Stewart, etc. Empirical examination of a collaborative web application. *In 2008 IEEE International Symposium on Workload Characterization*.

[5] Andreas Gal, etc. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *To be appeared in Programming Language Design and Implementation (PLDI) 2009*

[6] Pu, Calton, etc. An Observation-Based Approach to Performance Characterization of Distributed n-Tier Applications. *In 2007 IEEE International Symposium on Workload Characterization*.

[7] Jordan Nielson, etc. Benchmarking Modern Web Browsers.

[8] Firefox 3 Memory Benchmarks and Comparison http://dotnetperls.com/Content/Browser-Memory.aspx.

[9] JavaScript Performance Rundown http://ejohn.org/blog/javascript-performance-rundown/.

[10] Zimbra reaches milestone, reinforces Yahoo's Mail business http://blogs.zdnet.com/BTL/?p=14035

[11] Alan Grosskurth. Architecture and evolution of the modern web browser.

[12] Windmill website http://www.getwindmill.com/.

[13] Firefox 3.1b3 source code package http://releases.mozilla.org/pub/mozilla.org/firefox/releases/3.1b3/source/.

[14] Firefox Build environment 1.3 http://ftp.mozilla.org/pub/mozilla.org/mozilla/libraries/win32/.

[15] Chrome Source Code http://dev.chromium.org/developers/how-tos/get-the-code.

[16] Chrome Multi-process Architecture http://dev.chromium.org/developers/design-documents/multi-process-architecture

[17] Windmill architecture http://www.getwindmill.com/wp-content/uploads/2008/08/windmill_slide_2008_web.pdf.

[18] iSuppli: Quad-core chips included in nearly half of mainstream desktops by 2009 http://www.edn.com/article/CA6434456.html?industryid=48885

[19] Chrome Threading http://dev.chromium.org/developers/design-documents/threading

[20] How Chromium Displays Web Pages http://dev.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome.

[21] V8 JavaScript Engine Design Elements http://code.google.com/apis/v8/design.html