A Performance Study of Modern Web Applications

Ramesh Radhakrishnan and Lizy K. John
Laboratory for Computer Architecture
Electrical and Computer Engineering Department
The University of Texas at Austin
Austin, Texas 78712
Tel: 512 471 8011 Fax: 512 471 8967
{radhakri,ljohn}@ece.utexas.edu
#2 Performance Evaluation and Prediction

Abstract

Web server programs are one of the most popular computer applications in existence today. Our goal is to study the behavior of modern Web server application programs to understand how they interact with the underlying Web server, hardware and operating system (OS) environment. We monitor and evaluate the performance of the system using hardware performance counters, for different workloads and identify the bottlenecks in the system based on the analysis of the collected data. Our workloads include static requests (to HTML files and images) which form a major percentage of the web traffic, as well as dynamic requests in the form of CGI (Common Gateway Interface) scripts and Servlets, both of which are gaining popularity and hence emerging to be a significant part of today's Web traffic.

We include not only the Web server itself in our studies, but also the server applications that run under its control as CGI scripts or Java Servlets. An UltraSPARC-II is used to run the server program in the UNIX environment. The CPI for the dynamic workloads was observed to be approximately 20% higher than the static workloads. The major factor that we could attribute to this was external (L2) cache misses. Dynamic workloads were observed to have a higher I-Cache and D-Cache miss rate. I-Buffer stalls caused due to I-Cache misses and processor stalls due to D-Cache misses form 45% of all the stalls in the dynamic workload. On the other hand, branch related I-Buffer stalls and data miss stalls constitute 75% of all the stalls in the static workload. As expected the server workloads are operating system intensive.

1 Introduction

In the past few years the World Wide Web has experienced phenomenal growth. Not only are millions browsing the Web, but hundreds of new Web sites are added each day. Yet, despite the increasing number of Web servers in use, little is known about their performance characteristics. Server hardware, Web server software, and a connection to the Internet are required elements of any Web site. To generate the best possible performance for any Web site, an understanding of the interrelated effects of these three elements on Web server performance is vital. Increasing Web server performance means reducing the response time to service a request. This depends on a lot of factors involving the underlying hardware, server software, the interaction between the software and hardware, and the characteristics of the workload. Previous research in Web server performance [22] suggests better hardware system performance to be the most obvious way of improving Web server performance.

Previous studies have shown that Web servers spend around 85% of their cycles executing OS code [7], as compared to 9% by the SPEC95 suite[9]. This suggests that Web server software may well exhibit significantly different characteristics from standard benchmarks like SPEC95 [15]. This, in turn, raises questions about performance of Web servers on current microarchitectures, since many of the recent innovations in microarchitecture have been justified on the basis of increasing SPEC95 performance. Therefore it is important to understand the behavior of modern superscalar processors when running Web server workloads. This can lead to improved Web server performance by providing insights about microarchitectural design changes or making modifications in server software (to better utilize the hardware resources). Previous research has shown that CPU and memory can form a bottleneck for a heavily loaded server system [3]. However, microarchitectural studies based on Web workloads have not been previously done, to identify the exact nature of these bottlenecks. Efforts have been made to identify the impact of secondary cache size on the performance of the server, however this study was based on static workloads [20]. Our work uses the on-chip performance counters to observe the system performance while running web workloads. The observations made in this paper are based on actual measurement and therefore accurately represent the behavior of the system being measured.

In this paper, we study the behavior of a popular Web server running on a modern superscalar machine to understand the interaction between the Web server software and hardware. First, we compare Web server performance for static and dynamic workloads. We include dynamic requests to model a real-world workload, since current Web benchmarks do not address this issue. Dynamic requests are excluded in current Web server benchmarks [18] [17] [14], so these benchmarks are less representative of real world workloads. Second, we characterize the performance of a modern superscalar machine while running Web workloads. Previous studies on processor characterization have used technical and commercial workloads, however studies based on Web workloads are lacking.

The Apache web server [1] was chosen as the Web engine to service the static requests and CGI scripts. The Apache server is a widely used, stable Web server with good performance. For studying Servlets workload behavior, the Java Servlet Development Kit(JSDK) provided by Sun Microsystems is used to service the requests. We gather important measurements to understand the system behavior and identify bottlenecks, using the performance counters that are implemented in the hardware. Emphasis is placed on gathering data that would help in understanding the hardware and software performance for the different workloads.

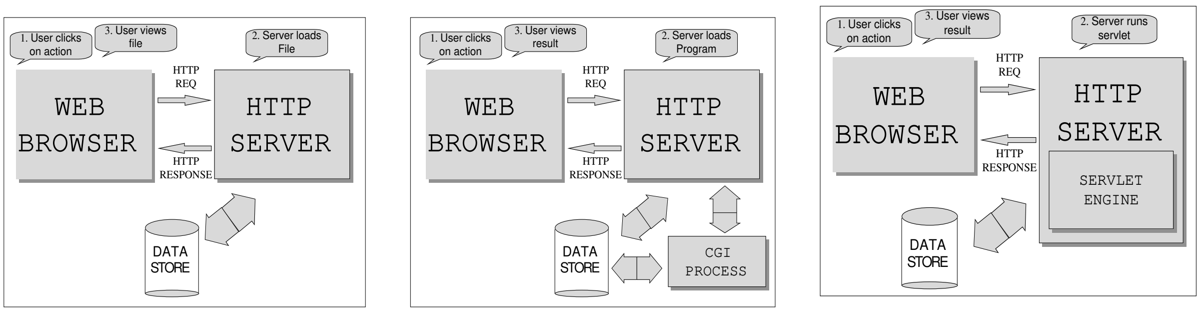
The remainder of this paper is organized as follows. In the next section we give a brief description of the operation of Web servers. In Section 3, we describe the characteristics of our experimental environment including the key features of the UltraSPARC microarchitecture. We define the metrics of interest in our study and justify our choices in Section 4. Section 4 also presents our data and interprets it. We suggest some future work and conclude in Section 5.

2 Background

2.1 Functioning of the Server for different Workloads

The World Wide Web (WWW) is a distributed hypertext based information system developed at CERN based on the client-server model [12]. A user requests and views information on the WWW using a browser. The browser forwards the request to the appropriate Web server, which responds with the required information. The function of a server is to process the request, load the appropriate file and send it back to the client. The request and response are exchanged over a TCP connection. The connection is terminated once the response is completed. This process is repeated for every document that a client requests.

Earlier, all the requests made by the client were static in nature, thus the function of the server was to load the appropriate file and send it to the client. However, the functionality of current servers have been extended to incorporate dynamic requests, the server can generate data or information based on the contents of the request. It does this by running certain programs on the server machine, generating results and then sending it back to the client. Although dynamic capabilities add to a Web server's overall capability, there is additional overhead involved in the creation of a new process and the subsequent interprocess communication generated. The impact of these these overheads are different for the various dynamic Web technologies, depending on their implementation.



(i) Static request (ii) CGI programs (iii) Java Servlet

Figure 1: Client-server model for different Web workloads

The difference in the working of the Web server while servicing static and dynamic requests can be understood better by looking at Figure 1. The figure shows a high level block diagram of the operation of the client-server model, for static and dynamic requests.

1. The first part shows the transaction path for a static request. In this case, the client makes a request, which is received by the Web server as an HTTP request. The server then loads the appropriate document and transmits it through the TCP connection over to the client. The connection is then closed. There is very little overhead involved in such a transaction. Requests for HTML files and images are usually static in nature.

2. The second part of the figure shows a dynamic request, which is processed using CGI [8]. The user makes a request, and the server starts a process upon receiving the request. The

	UltraSPARC System
Model	Sun Ultra5 (SPARC station 5)
Processor	UltraSPARC-II
Memory Size	128 MB
Cache	Split Instruction and Data Cache 16KB, 2-way set associative I Cache 32 byte line size 16KB direct-mapped D Cache, write-through 16-byte sub-blocks per cache line Unified External Cache (Level 2 or L2)
Instruction Issue	Execution of up to 4 instructions per cycle
Instruction Completion	Limited out of order execution
Branch Prediction	Dynamic branch prediction scheme implemented in hardware, based on two-bit history

Table 1: The UltraSPARC-II Processor Characteristics

process can be a script file, an executable or any number of such processes depending on the request. Thus, CGI enables the server to create information on the fly, which is sent back to the client. CGI programs are executed in real time, to output dynamic information.

3. The third part of the figure, explains a relatively new technology, Java Servlets [6]. Servlets are the Java equivalent of CGI programs. To output dynamic information, the server executes Java programs which are run on the JVM. Hence the JVM has to be invoked to run Servlets (resulting in additional overhead) and the generated information is sent back to the client.

3 Experimental Setup

3.1 The UltraSPARC-II Microarchitecture

The hardware platform used to characterize the web server running in a UNIX environment was the UltraSPARC-II. We give a brief description of the UltraSPARC-II architecture in Table 1. The UltraSPARC-II is a high-performance, superscalar processor implementing a 64-bit RISC architecture [19]. It is capable of sustaining the execution of up to four instructions per cycle. This is supported by a decoupled prefetch and dispatch unit with an instruction buffer. Instructions predicted to be executed are issued in program order to multiple functional units and are executed in parallel. Limited out of order execution is supported for these predictively-issued instructions. Long latency floating point and graphic instructions may complete out of order with respect to independent instructions.

3.1.1 Performance Monitoring Counters

The UltraSPARC-II processor provides two hardware counters to monitor the performance of programs. Up to two performance events can be measured simultaneously. The events to be monitored can be specified through the Performance Control Register (PCR) to a pair of 32-bit Performance Instrumentation Counters (PICs). Access to the PCR is privileged and non-privileged accesses to the PICs can be restricted. The two 32 bit PICs accumulates over 4 billion events before wrapping around. Additionally, the UltraSPARC implements a 63-bit TICK counter, which is used to measure the clock cycles elapsed. These four performance registers (PCR, TICK and the two PICs) were used to collect statistics for important events in the processor running under the different workloads. The performance counters could be accessed to measure performance using the tool Perf-monitor [13] which allows one to set and read the performance counters. A description of the events that the counters can be used to measure is given in Table 2. More detailed information about the counters and hardware registers mentioned here can be obtained from [21].

Event id	**Description**
Cycle_cnt	Accumulated cycles
Instr_cnt	The number of instructions completed
Dispatch0_IC_miss	cycles I-buffer is empty from I-Cache miss. Cycles include an E-Cache miss processing in case of E-Cache miss.
Dispatch0_mispred	cycles the I-buffer is empty from branch misprediction. The total number of pipeline bubbles is approximately twice this number.
Dispatch0_storeBuf	cycles that the first instruction in a group is a Store, and a the store buffer cannot hold additional stores
Load_use	cycles stalled when there is a load in the execute stage which depends on an earlier load result, currently unavailable.
Load_use_RAW	cycles that there is a load in the execute stage, and it has a read-after-write dependency on the oldest outstanding load.
IC_ref	I-Cache references
IC_hit	I-Cache hits
DC_rd	D-Cache read references
DC_rd_hit	D-Cache read hits
DC_wr	D-Cache write references
DC_wr_hit	D-Cache write hits
EC_ref	E-Cache references
EC_hit	E-Cache hits

Table 2: Events measured using the on-chip counters on the UltraSPARC-II Processor and associated Performance Metrics

3.2 Software environment and Web Server Engine

The main Web server engine used in our study is Apache 1.3.0 [1]. The Apache Web server is widely used, has good performance and is non-proprietary. Apache has built-in capability for handling CGI requests, and can be extended to run Servlets using third-party modules. Apache was run on an UltraSPARC-II and measurements were made using the performance counters that are provided in the processor. We use the Solaris 2.6 Operating System as the software platform to run the server program. In addition to Apache, we use the web server engine *servletrunner* provided with JSDK 2.0. to handle requests for Java Servlets. This package is provided by Sun Microsystems to help develop and test Servlets before using them on a commercial or more stable, "Java enabled" Web server. The server engine provided in the JSDK may not be optimized for performance since it is only used for testing Servlets. We are currently working on trying to add Servlet support to Apache, by using third-party add-on modules, which could result in better performance results for Servlets than what we see currently in this study. The Java Virtual Machine (JVM) that executes the Servlets is the Sun Java JDK 1.6.

Web Server Engine	Requested File	Nature of request	File size
Apache 3.0	index.html	static	1622 bytes
Apache 3.0	apache_pb.gif	static	2326 bytes
Apache 3.0	printenv	dynamic	126 bytes
JSDK 2.0	SnoopServlet.class	dynamic	1107 bytes

Table 3: Details of the requests to the server on the different platforms

The static workload consisted of requests made for an HTML page containing text and a request for an image (GIF file). The dynamic requests consisted of a call to an executable invoked through a CGI interface. A request made for a Java Servlet and formed the other half of the dynamic workload.

A dummy client, running on the same machine as the server makes requests. This would minimize any latencies that occur on the network. Since we can specify which processes we want to monitor, we only gather data for the web server deamons and the data collected is not affected by the client program. The details of the different requests made by the dummy client and measured using the counters are given in Table 3. The static workload is formed by repeatedly requesting *index.html*, an HTML document, and *apache_pb.gif*, a GIF image. The Perl executable *printenv*, which is invoked using a CGI interface and *SnoopServlet.class*, a Java class file, form the dynamic workload. This Servlet posts information about the server environment by executing the program in the interpreted Java environment. The client program would establish a connection and make 20 requests for each file. A series of such runs are made in order to keep the server busy and heavily loaded. The latency in servicing these requests is measured using the hardware counters. This particular methodology is obviously contrived, but it is simple enough to repeat extensively and serves as a starting point for characterizing how emerging web applications behave. The main problem that can arise with the methodology we used is that for static requests the files could be cached in the file system cache and the numbers might not reflect the true performance of a real web server. To avoid this we used multiple static http files and images of the same size to mitigate this effect. For the dynamic requests the server has to run the application each time and generate the results and therefore we do not have to address this problem.

4 Performance Evaluation

The on-chip counters were used to gather statistics that would help us understand the performance of the Web server, running under different workloads. The events measured are chosen to explain the performance numbers obtained, while testing the performance using the experimental setup described earlier. The interesting statistics obtained under the UNIX platform and our observations based in the data collected are described here.

4.1 Metrics Measured and the Validation Process

Important statistics like cycle count, instructions executed, cycle stalls, and cache information are accessed using the performance counters. Different modes can be used to measure system events only, user events only or both together. In this study we include the numbers for both system

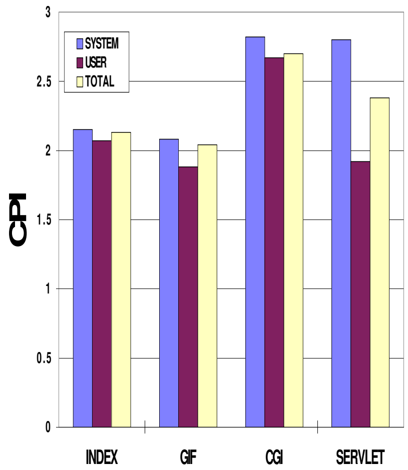


Figure 2: CPI for different workloads

events and user events, since it is interesting to see what percentage of the cycles are spent in the OS kernel and how this has an affect on performance.

To validate the different runs and confirm that context switches and other processes were not measured, we compare each run using the free running TICK counter available on the SPARC. The TICK counter increments on each clock cycle, and we can check to see if the performance counters measured any other processes because of a context switch. This allows us to minimize errors due to context switches and also use information from comparable runs. Since the runs are never exactly the same, we only took the values obtained from runs that showed a variance of less than 5%. Furthermore, we look at the log files generated by Apache to see that the proper number of requests were made and completed successfully. Each entry in the log file would give information about the request made and whether it was completed without error.

We measured the instruction buffer performance to see how factors like instruction cache misses and branch mispredictions were affecting the processor performance. The performance of the store buffer and stalls caused by various factors was also measured using the counters.

4.2 Performance of different workloads

Cycles Per Instruction (CPI) for the different workloads are presented in Figure 2. It is observed that the static workload has a lower CPI than the dynamic workload. Table 4 presents the instruction count and cycles executed for the different workloads. The server executes a higher number of instructions for the Servlet requests, because the JVM is invoked to run the Servlet, which results in additional instructions being executed.

The first interesting observation from Table 4 is that the number of instructions executed are comparable for CGI and static workloads. Even though the server has to create separate processes to handle the dynamic requests, we see that the number is very high only for Servlet. This is because the server will invoke the Java Virtual Machine (JVM) to run the Servlet class file. This causes additional instructions being executed and is reflected in the instruction count.

As expected, the CPI is higher while executing dynamic requests. This is because new processes have to be created, resulting in context switches and additional overhead. The overhead for creating new processes and calling across these process boundaries deteriorates the performance. This effect

Static workload	Instr	Cycles	CPI
index(system)	152159904	327325408	2.15
index(user)	11424319	23748612	2.07
index(both)	159348832	350395584	2.13
gif(system)	208139824	433525344	2.08
gif(user)	10934707	21821480	1.88
gif(both)	207659920	442459808	2.04
Dynamic workload	Instr	Cycles	CPI
cgi(system)	141329088	399676160	2.82
cgi(user)	30277060	80925312	2.67
cgi(both)	178319120	482511360	2.70
servlet(system)	151229216	424606304	2.80
servlet(user)	140923376	271369728	1.92
servlet(both)	287816608	686206592	2.38

Table 4: Breakdown of instructions, cycles and CPI

is seen in the case of a CGI request, where the CPI is much higher than the static workload although the instructions executed are comparable. In the case of Java Servlets, the CPI is higher than the static workload. However, since the Web server engine used to run this workload, *servletrunner*, is written in Java, it runs the Servlet on the JVM. A higher CPI is observed because the Java environment is interpreted, leading to poorer performance as compared to running compiled scripts. Also, *servletrunner* is not a commercial server: its purpose is to test Servlets before they are deployed in a Web server. Hence, it is not optimized for performance, which may be another reason contributing to its high CPI.

Another observation from Table 4 is that a high percentage of cycles are spent in the system mode as compared to the user mode. It is seen that *index* and *cgi* spend more than 80% of cycles executing system code. For Servlets the percentage is lower, around 61%, but is still significant compared to other standard benchmark programs used for measuring system performance. The reason why the Servlet workload shows lower percentage of system cycles is because *servletrunner* does not have the full functionality of a commercial Web server. For instance, there is no logging tasks performed, and no security measures provided which would result in higher cycles spent executing OS code. The high percentage of cycles spent in the system mode for a Web server indicates that, in order to optimize server performance we should concentrate on providing more OS support or at least optimize the OS code for those system calls that were made frequently. Research done by Gaurav et. al. [5] shows how adding OS support can increase the performance significantly.

4.3 Analysis of Pipeline Stalls and Cache Misses

We use the performance counters to measure pipeline stalls and cache misses, which could be factors leading to the higher CPI observed for the server while running the dynamic workload.

Cache Performance Good cache performance is important in achieving optimum performance. We calculate the cache miss rates across the different workloads. Instruction cache misses, data cache misses and external cache misses were calculated, and were observed to correspond with the CPI calculated. Figure 3 gives miss rates that were observed for all the three caches. The miss rates for the external cache were similar for the different workloads, with the exception of Servlets which showed a lower external cache miss rate. Static workloads were observed to have lower miss rates in both the instruction and data cache, as compared to dynamic workloads. Surprisingly, the external cache was observed to show a high miss rate for all workloads. The miss rates for the external cache ranged from 13.47% for Servlets to 17.89% for the CGI programs.

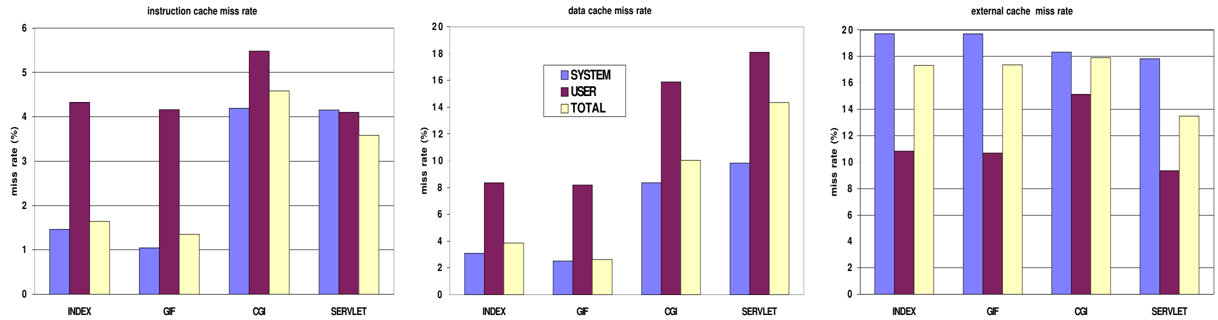


Figure 3: Miss rates for instruction, data and external caches

Since the dynamic workloads spawn a new process for handling the request, it brings a new working set into the instruction cache, resulting in higher miss rates. The higher miss rates seen for the dynamic workload, would lead to a higher number of pipeline stalls, as seen in the next section. The miss rates for instruction caches are higher by a factor of 4 for the dynamic workloads and this proves to be expensive. The instruction cache miss rates for static workloads range from 1.35% to 1.64%, and from 3.58% to 4.58% in the case of dynamic workloads. The difference in data cache miss rates are also seen to be greater by a factor of 4 for the dynamic workload. Data cache miss rates for the static workload range from 2.62% to 3.84% , and from 10.03% to 14.34% for the dynamic workload.

There is a high correlation that is observed between the instruction cache and data cache miss rates, to the CPI that was measured. The dynamic workloads incur higher cache misses due to the execution of other processes, created to provide additional functionality to the server.

Pipeline stalls Table 5 gives details about the stalls that occurred in the processor pipeline due to store buffer full conditions, data waiting and stalls in the instruction buffer. The table lists the percentage of total clock cycles that the flag associated with a stall was seen to be true. From this we can estimate the total number of stalls that were caused by each of the different factors. We describe what each column in Table 5 represents below, so that the data can be interpreted from the Table more easily.

- `I-Buffer Stalls:` This is further divided into IC miss and branch.

  `IC miss:` lists the percentage of cycles that the I-buffer is empty from an I-Cache miss. This also includes E-Cache miss processing if an E-Cache miss also occurs.

  `branch:` lists the percentage of cycles the I-buffer is empty from branch misprediction. Since branch misprediction invalidates any instruction issued after the current instruction, the total number of pipeline bubbles is approximately twice as big as measured from this count.

- `Store buffer full:` This column gives the percentage of cycles that the first instruction in the group is a store, and the Store buffer can not hold additional stores.

Different Workloads	I-Buffer Stalls		Store buffer full	Load Stall	Load RAW
	IC miss	branch			
index	8.64	10.72	0.62	16.09	0.65
gif	6.91	13.12	0.49	19.02	0.49
cgi	17.98	6.76	2.47	22.70	1.10
servlet	17.77	5.89	2.97	20.77	0.94

Table 5: Breakdown of stalls in the processor (% of Total cycles)

- `Load Stall`: This is the percentage of cycles that an instruction in the execute depends on an earlier load result that is not yet available.
- `Load RAW`: This gives the percentage of cycles that there is a load in the execute stage and there is a read-after-write hazard on the oldest outstanding load. This basically means that the load data is being delayed by completion of an earlier store.

Instruction buffer and store buffer stalls cause the processor to stall in the dispatch stage of the pipeline. We see alternate behavior here for the different workloads. The static workload shows lower instruction buffer stalls due to an IC miss, as compared to the dynamic workload. However, instruction buffer stalls caused by branch misprediction is higher for static workloads as compared to the dynamic workload. The branch stalls are least for the Servlet, which runs in an interpreted environment and therefore shows better branch behavior.

Stalls in the dispatch stage of the pipeline due to store buffer full conditions were higher for dynamic workloads. The percent of stalls are higher by a factor of approximately 5 times. Similarly a higher percentage of stalls were observed in the execute and grouping[1] stages of the pipeline for dynamic workloads. These stalls are caused by the processor not being able to execute an instruction because it is waiting for data from a previous load instruction. These stalls are seen to be high for both static and dynamic workloads.

I-Buffer stalls due to I-Cache misses and stalls due to D-Cache misses contribute to more than 45% of all the stalls in the dynamic workloads. Stalls in the static workload is dominated by branch related I-Buffer stalls and stalls for data (load miss) which together account for more than 75% of the all the stalls. Since many of these stalls can overlap, it is extremely difficult to precisely correlate the various stalls with observed CPI. Ding and Bhandarkar also made a similar comment in their studies on the Pentium Pro [4].

5 Conclusions

We studied the behavior of a modern Web server to understand the interaction with the underlying hardware and operating system. We used hardware performance counters on the UltraSPARC-II to monitor performance. Hardware monitoring is a non-invasive method for studying the behavior of a system and has very low overhead. The behavior of the server program was characterized while servicing static and dynamic requests. The major observations are as follows:

[1]In this stage a group of up to four instructions (two integer and two floating-point/graphics), are assembled and dispatched in one cycle.

- It was found that the additional functionality provided by the dynamic behavior of the server is achieved at the cost of a high overhead. The CPI for dynamic workloads were approximately 22.8% higher than the static workloads.

- Dynamic workloads resulted in higher cache misses as compared to the static workloads. The I-Cache misses for static workloads ranged from 1.3% to 1.6%. The I-Cache misses for the dynamic workloads were higher, ranging from 3.6% to 4.5%. We see a similar observation for D-Cache misses also, 2.5% to 4% was the miss rate observed for the static workloads and the dynamic workloads had a higher miss rate of 10% to 14.2%.

- I-Buffer stalls due to I-Cache misses and stalls due to D-Cache misses contribute to more than 45% of the stalls in the dynamic worklods. Stalls in the static workload is mainly dominated by branch related I-Buffer stalls and stalls for data (load miss) which together account for more than 75% of the stalls.

The server code was seen to spend more than 80% of its cycles executing system code. Benchmark suites like SPEC95 are seen to spend less than 10% of cycles executing system code, hence further studies on modern processor characterization using Web workloads is a significant research area.

Our work is only a starting point in measuring the dynamic capability of a Web server. We plan to measure behavior of several other Web servers. A Java Web server, like Jigsaw [10], should show improvement in the execution of Servlets, since the Servlets would run in the same JVM used for the server. The impact of using different CGI programs, written in different languages or scripts, should provide a better understanding of CGI performance. Evaluating the performance of other technologies that provide dynamic capabilities to current Web servers is another area for further research. We are in the process of collecting data for longer runs, varying the file size, distribution, and frequencies to study its impact on the hardware. Including dynamic requests to existing benchmarks is important to model real-world workloads accurately. Further studies on Web workload characterization, taking into account the emerging technologies, is needed to create better benchmarks.

References

[1] The Apache HTTP Server Project, `www.apache.org`

[2] M. Arlitt and C. Williamson, "Web Server Workload Characterization: The Search for Invariants", *Proc. of the 1996 ACM SIGMETRICS Conference on the Measurement and Modelling of Computer Systems*, pp. 126-137, 1996.

[3] J. Almedia, V. Almedia and D. Yates, "Measuring the behavior of a World-Wide Web server", in *Proc. 7th Conf. High Performance Networking*, pp. 57-72, April 1997.

[4] Dilip Bhandarkar and Jason Ding, "Performance Characterization of the PentiumPro Processor", *Proc. of the International Conference on High Performance Computer Architecture*, 1997.

[5] Gaurav Banga, Peter Druschel and Jeff Mogul, "Resource containers: A new facility for resource management in server systems", to appear in the *Proc. of the Third Symposium on Operating System Design and Implementation*, New Orleans, LA, February 1999.

[6] Phil Inje Chang, "Inside The Java Web Server, An Overview of Java Web Server 1.0, Java Servlets, and the JavaServer Architecture", White Paper, Sun Microsystems, `http://java.sun.com/features/1997/aug/jws1.html`

[7] B. Chen, Y. Endo, K. Chan, D. Mazieres, A.Dias, M. Seltzer and M. Smith, "The Measured Performance of Personal Computer Operating Systems.", *Proc. of the Fifteenth Symposium on Operating System Principles*, pp. 299-313, 1995.

[8] M. Felton, CGI: Internet Programming With C++ and C, Prentice Hall Press, 1997.

[9] N. Gloy, C. Young, J. Chen, and M. Smith, "An Analysis of Dynamic Branch Prediction Schemes on System Workloads", *Proc. of the International Symposium on Computer Architecture*, May 1996.

[10] Jigsaw, `http://www.w3.org/Jigsaw/`

[11] B. Lauri and P. Lauri, "Apache, The Definitive Guide", O'Reilly and Associates, 1997.

[12] T. Berners-Lee, R. Calliau, A. Luotonen, H. Nielsen and A. Secret, "The World-Wide Web", *Communications of the ACM*, 37(8), pp. 76-82, August 1993.

[13] Magnus Christensson, "Perf-Monitor, A Package for Measuring Low-Level Behavior on the UltraSPARC", Swedish Institute of Computer Science.

[14] NetPerf, `http://www.cup.hp.com/netperf/NetperfPage.html`

[15] The Standard Performance Evaluation Corporation, `www.spec.org`

[16] The Workload for the SPECweb96 Benchmark, `http://www.spec.org/osg/web96/workload.html`

[17] SPECweb96 Benchmark,
`http://www.spec.org/osg/web96/`

[18] G. Trent and M. Sake, "WebStone: The First Generation in HTTP Server Benchmarking", White Paper, Silicon Graphics, February 1995.

[19] K. Normoyle, M. Csoppenzky, A. Tzeng, T. Johnson, C. Furman, and J. Mostoufi, " UltraSPARC-IIi: Expanding the Boundaries of a System on a Chip", IEEE Micro, Vol. 18, No. 2, March/April 1998.

[20] Roy Saharoy, "Performance Impact of Secondary Cache Size on Web Servers", *Second Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Held in Conjunction with HPCA-5, 1999.

[21] The UltraSPARC User's Manual, `http://www.sun.com/microelectronics/UltraSPARC-II/`.

[22] Measuring Web Server Performance, SunWorld, September 1997,
`http://sunsite.uniandes.edu.co/sunworldonline/swol-09-1997/swol-09-webserver.html`