Workload Characterization of Multithreaded Java Servers

Yue Luo and Lizy K. John

Laboratory for Computer Architecture
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin,TX 78712
{luo,ljohn}@ece.utexas.edu

Abstract

Born as a browser plug-in, Java has now gained substantial popularity in the commercial server arena. Researchers have studied database servers and web servers as well as Java client benchmarks, but the characteristics of Java server applications are not well understood. A Java server needs to manage a large number of threads in order to support many simultaneous client connections. The impact of the large number of threads on processor microarchitecture has not been studied in the past.

In this research, we characterize the behavior of two Java server benchmarks, VolanoMark and SPECjbb2000, on a modern superscalar system with a state-of-the-art Java Virtual Machine. We compare Java server applications with SPECint2000 and also investigate the impact of multithreading by increasing the number of clients. Java servers show poor instruction access behavior, including high instruction miss rate, high ITLB miss rate, high BTB miss rate and, as a result, high I-stream stalls. With increasing number of threads, the instruction behavior improves, suggesting code sharing and constructive interference. All the aforementioned miss rates as well as the branch misprediction rates drop due to the constructive interference among threads. But resource stalls increase and eventually dwarf the diminishing I-stream stalls. The most significant hindrance to scalability with more connections/threads is the phenomenon that the number of instructions needed for a unit of effective work increases as more clients are connected to the server.

1. Introduction

Java was initially targeted for web clients, but now it has emerged as a competitive paradigm for server platforms because of the superb property of "write once run everywhere" and the enhanced security of Java. Typically, Java execution suffers from software translation overhead, but the overhead of Just-In-Time (JIT) compilation can easily be amortized in long-running server applications. Moreover, Java provides unique opportunity for dynamic optimization, which will further boost server performance.

Server workloads differ significantly from those of the clients. Therefore, not surprisingly, performance characteristics unveiled by studying client workloads may not be applicable to server side. One of the major distinctive characteristics of server applications is that they usually need to support many concurrent client connections. Traditionally, there are three techniques to handle concurrent connections, i.e. I/O multiplexing, polling and signals. Unfortunately, in current Java, none of these techniques is directly available. To compensate for the lack of these features, a Java developer usually creates one or more separate threads to manage each client connection [11]. Therefore performance under the condition of a large number of threads is crucial for a Java server to support multiple clients simultaneously. A solution is being proposed for this problem in JDK1.4 beta [17].

The number of threads, and thus the number of simultaneous clients that a particular system can support is generally constrained by the system resources and the resource requirement of each thread. Optimization to maximize the number of threads is out of the scope of this paper. The goal of this paper is to provide a detailed characterization of the impact of multithreaded Java server application on processor microarchitecture. For this purpose, we perform two types of experiments: First we compare Java server benchmarks with SPECint 2000, a more "traditional" workload, to find out the characteristics of the Java server applications. Then we

increase the number of simultaneous threads of the Java server benchmarks to determine how multithreading would impact the processor microarchitecture.

The rest of the paper is structured as follows. In next section we introduce some related work. Section 3 provides our experimental methodology as well as some background on the benchmarks we use. Section 4 and Section 5 present our main results for the two types of experiments. Section 6 summarizes key observations and offers concluding remarks.

2. Related Work

As commercial applications are becoming more and more important, there have been ongoing efforts to characterize the performance of commercial workloads [2, 10, 8, 5, 14, 1]. Majority of these studies have been focused on OLTP and DSS applications as well as on web servers, which are highly optimized and well-established applications developed in C or C++ and compiled into native machine binaries.

Java has also been hot in research for years. Most of the results of Java workload characterization are based on the study of SPECjvm98 [16, 9, 12], which is a client type benchmark suite. SPECjvm98 is found to show as high as 31% kernel activity, most of which is *utlb*, a TLB service routine. This indicates a high TLB miss rate. These applications exhibit poor ILP and are insensitive to wider issue width [9]. However, they show better instruction cache performance than C/C++ applications [12]. At client level, software translation is seen to dominate, however, we do not expect that at server level.

Commercial pure Java servers are just emerging and so are the research efforts in quantifying their behavior, especially the multithreading impact. Cain and Rajwar [4] studied the effect of multithreading on branch prediction and cache behavior in SPECjbb2000 and TPC-W by full system simulation of a coarse-grained multithreaded processor. In this research they found

destructive interference between threads. Some previous research, however, shows constructive interference between threads in multithreading. Hily and Seznec [6] studied the behavior of branch prediction while simultaneously executing several threads of instructions. It was observed that some branch prediction algorithms do benefit slightly from multithreading within a program. The conflicting results on the impact of multithreading necessitates more studies with multithreaded workloads.

In this paper we present our study of two Java server benchmarks on a real machine with a modern processor and a state-of-the-art JVM. The objective is to understand the behavior of Java servers, particularly multithreading, based on actual execution rather than simulation. The impact of large number of threads on caches, TLBs, branch predictors, etc is investigated.

3. Methodology

3.1 Benchmarks

In this paper, we use the VolanoMark benchmark and SPECjbb2000 to study multithreaded Java servers.

VolanoMark [18] is a pure Java server benchmark with long-lasting network connections and high thread counts. It can be divided into two parts, server and client, though they are provided in one package. The server is a slightly modified commercial chat server, VolanoChat, which accepts connections from the chat client (Figure 1). The chat client simulates many chatting users: it creates a number of chat rooms, continuously sends messages to server and waits for the server to broadcast the messages to all the users in the same chat room. VolanoMark can be run to test both speed and scalability of a system. In speed test, it is run in a loopback fashion with server and client on a single machine. In scalability test, the server and client are run on two

separate machines with high-speed network connections. VolanoMark server creates two threads for each client connection.

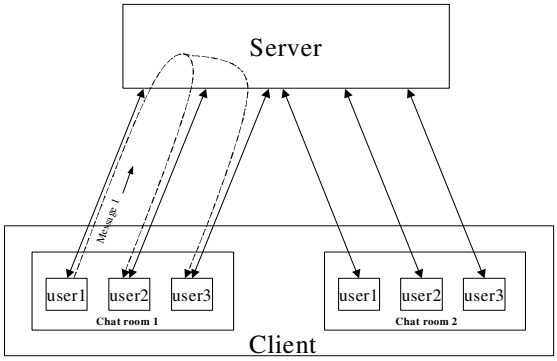


Figure 1 VolanoMark

SPECjbb2000 (Java Business Benchmark) [19] is SPEC's first benchmark for evaluating the performance of server-side Java. The whole benchmark is implemented within a single JVM. It is a Java program emulating a three-tier client/server system with emphasis on the middle tier (Figure 2). The emulation of the other tiers isolates the middle tier and simplifies the benchmark by not requiring user emulation or a database. (The implication is that combining a JVM with a high SPECjbb2000 throughput and a database tuned for online transaction processing will provide a business with a fast and robust multi-tier environment.) SPECjbb2000, like TPC-C, models a wholesale company with warehouses serving a number of districts. Customers initiate a set of operations such as placing new orders or requesting the status of an existing order. Additional operations are generated within the company, such as processing orders for delivery, entering customer payments, and checking stock levels. SPECjbb2000 assigns one active customer per warehouse, which is a data set of 25MB stored in binary trees. SPECjbb2000 is memory resident without inherent disk I/O. One warehouse maps directly to one Java thread. As the number of warehouses increases during the full benchmark run, so does the number of threads.

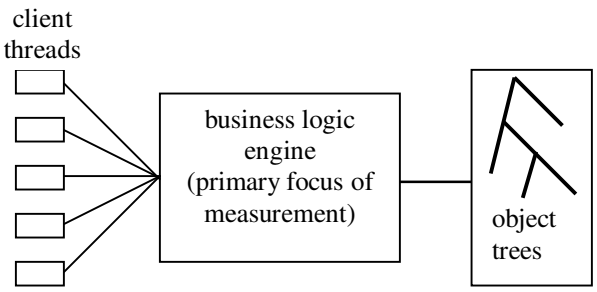


Figure 2 SPECjbb2000

To compare the two benchmarks to more "traditional" applications, we also measured SPECint2000 on the same platform. From the benchmark suite we chose 5 applications: 255.vortex, 300.twolf, 176.gcc, 252.eon and 186.crafty, which cover small, medium and big applications as well as a game playing program and the only program that is written in C++. SPECint2000 comprises such a wide variety of applications that we did not expect Java server application to stand out in terms of microarchitecture metrics. We do find, however, that Java servers have some unique properties.

3.2 Platform

Our monitoring experiments were performed on a Dell Precision 410 PC with one Pentium III processor and 1 GB of physical memory. The Pentium III is a superscalar out-of-order machine capable of issuing up to 5 uops and retiring up to 3 uops in one cycle. The processor has a 40-entry reorder buffer to facilitate retirement of instructions in order. The processor employs speculative execution using a two level branch predictor and a 512-entry branch target buffer (BTB). The processor has separate L1 data cache and L1 instruction cache. Each cache is 16 Kbytes in size, 4-way set associative with 32-byte block size, and employs LRU replacement algorithm. The data cache is write-allocate, non-blocking and dual-ported. The processor also has a unified 512 Kbyte 4-way set associative non-blocking L2 cache with a 32-byte block size.

The operating system on the system under measurement is Windows NT Workstation service pack 6a. We use the Sun JDK 1.3.0 with Hotspot Server (build 2.0fcs-E, mixed mode).

3.3 Monitoring Method

We use Pmon [15], a tool to read the on-chip performance monitoring counters of Pentium III, to monitor the application. Pmon consists of two parts, a device driver and a control program. The driver reads the performance counters [3,7] of Pentium III processor while the control program controls the measurement process and logs the results. Since we developed the whole tool ourselves, we have better control over it than any other performance counter tools like Intel's P6Perf. The overhead of Pmon is extremely small because it does not have GUI displays and does not write results to disks during measurements. Thus the tool incurs no disk I/O activity given enough memory. The low overhead associated with the tool allows us to perform the measurements in a non-invasive fashion. The operation of the tool was verified by several test cases and by comparing with VTune and P6Perf.

Our experiment with VolanoMark follows a procedure similar to that of the scalability test. We run VolanMark client on a different machine from the server. Each chat room has 20 users in it, a default value in VolanoMark. We vary the number of chat rooms from 1 to 40 resulting in a connection number range of 20 to 800. We measure only the server activity. SPECjbb2000 is a data intensive application with 25M bytes data for each warehouse/thread. The maximal number of threads that our system can afford without significant page faults is 25. Therefore, we increase the number of warehouses from 1 to 25 in our experiments.

Server applications have different performance implications compared to standalone client applications. Though it is desirable to have quick starting and shutdown processes, the most important aspect of server performance is how it responds to client requests. To synchronize our

measurements with client connections in VolanoMark test, we add a wrapper to the client program. The wrapper sets up an extra connection to the server to trigger Pmon immediately before it starts the actual client. Pmon ends the measurement as soon as the wrapper closes the extra connection, which signals the end of the client program. In this way, we only measure connection creation, message transmission and connection closure and skip the starting and shutdown of the server. To avoid counter overflow, we chose a conservative 3-second period as the counter reading interval. Since SPECjbb2000 does not have a separate client program, it is impossible to isolate the server transaction activity from data initialization and report generation without instrumentation of the benchmark itself. The benchmark program has the ability to measure itself for reporting benchmark scores. We modify `Company.java` file to send signals to Pmon so that our measurement is synchronized with the benchmark's own measurement interval. To minimize the effect of instrumentation we only recompile `Company.java` and leave all other class files untouched. As can be seen from our measuring method, the JIT compiling part should be negligible in the results because JIT compiler is usually invoked in the starting phase, which is excluded from our measurement.

4. Comparing Java server benchmarks and SPECint2000

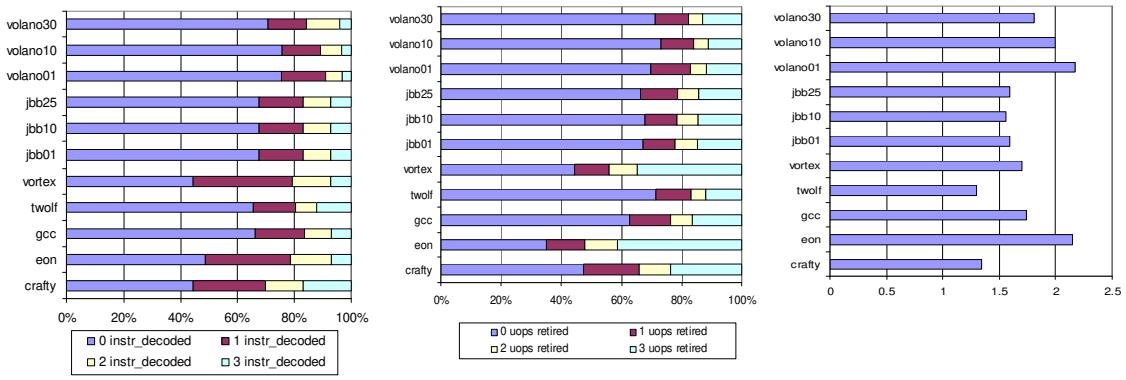
In this section, we compare VolanoMarks and SPECjbb2000 with SPECint2000. We run VolanoMark with 1, 10, and 30 chat rooms (the number of connection threads is 40, 400, and 1200, shown in figures and tables as `volano01`, `volano10` and `volano30`), and run SPECjbb2000 with 1, 10 and 25 warehouses (the number of warehouse threads is 1, 10 and 25, shown in figures and tables as `jbb01`, `jbb10` and `jbb25`). The microarchitectural parameters measured are similar to those measured by Bhandarkar et. al [3].

Benchmark	OS Cycle Percent	CPI	Data References per Instruction	Memory Transactions per 1000 Instructions
volano30	65.32%	3.03	0.620	10.671
volano10	58.25%	3.72	0.670	18.393
volano01	57.86%	3.68	0.722	9.514
jbb25	0.67%	2.31	0.572	15.972
jbb10	0.63%	2.29	0.578	14.142
jbb01	0.57%	2.28	0.593	13.143
vortex	1.06%	1.27	0.674	2.802
twolf	0.37%	2.25	0.593	17.224
gcc	1.04%	2.25	0.629	11.358
eon	0.24%	1.36	0.839	0.010
crafty	0.27%	1.22	0.499	0.346

Table 1 Java Servers vs. SPECint2000

Table 1 shows the percentage of cycles spent in kernel mode. SPECjbb2000 has neither file accesses nor network connections. And since it is a memory resident java database program, few page faults occur. Therefore, OS cycle time constitutes less than 0.7% of the total execution time, which is not very different from SPECint2000. VolanoMark, on the contrary, has significant OS part because it spends most of the time in receiving and sending network messages, which is mainly the task of the operating system. The number of threads is also large. Thus, scheduling and synchronizing these threads constitutes a major task of the operating system. Adding to this is the relatively simple operation of distributing messages in user code. Consequently, more than half of the execution time of VolanoMark is in kernel mode.

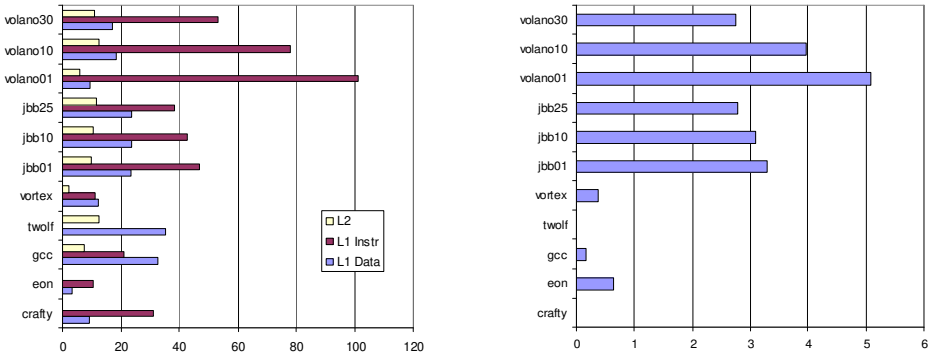
In table 1 Java server benchmarks show higher CPI than most of SPECint2000 programs. Operating system code is known to have worse CPI than user code [8]. With a much higher OS part, VolanoMark shows the highest CPI among all these benchmarks.



(a) Instruction Decode Profile **(b) Instruction Retirement Profile** **(c) Number of micro-ops per Instruction**

Figure 3 Instruction Decode/Retirement Behavior

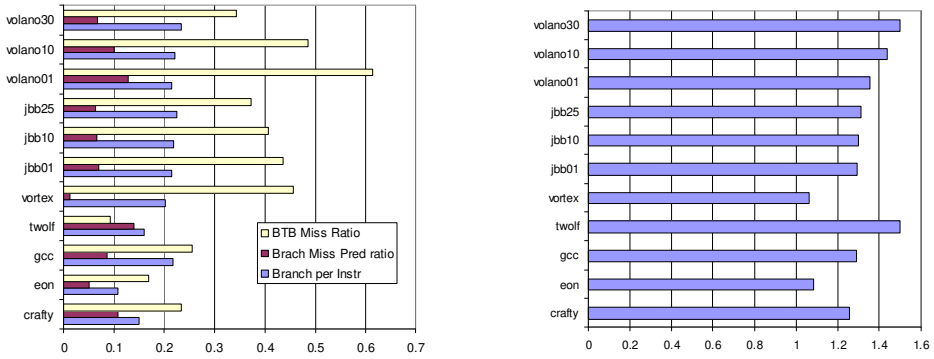
Pentium III has a microarchitecture similar to the Pentium Pro. Most instructions are converted directly into single uops. Some are decoded into one to four uops, and the more complex instructions require microcode. The processor has 3 decoders that can handle up to 3 instructions every cycle. Up to 5 uops can be issued every clock cycle to the various execution units and up to 3 uops can be retired every cycle [3]. But in Java servers, instruction level parallelism is limited: most of the time (more than 60%), the decoders are idle when executing Java server benchmarks (Figure 3a) and no uops can be retired in more than 60% cycles (Figure 3b). Only `twolf` and `gcc` in SPECint2000 show similar behavior. Because instructions in `twolf` translate to simpler uops than instructions in Java server benchmarks (Figure 3c), `twolf` still shows better CPI than Java servers (Table 1).



(a) Cache Misses per 1000 Instructions **(b) ITLB Misses per 1000 Instructions**

Figure 4 Cache and ITLB Behavior

Table 1 shows the number of data references per instruction and the number of memory transactions per thousand instructions. On average, Java server programs show a data reference rate similar to that of the SPECint2000. But Java servers have the worst overall cache behavior (Figure 4a). As might be expected, there is a strong correlation between L2 cache misses and memory transactions (Table 1). The most significant effect in Figure 4a is that Java servers have much higher instruction miss rate. High instruction cache miss rates have also been observed in traditional commercial server applications written in C or C++ [1, 2]. Server applications are complex and usually have large instruction footprint. Dynamically compiled code for consecutively invoked methods may not be located in contiguous address spaces. Thus the instruction locality is poor. And the OS code, which is the major part in VolanoMark, incurs more instruction cache misses when number of connections is small (Figure 7a). This causes VolanoMark to show very poor instruction cache behavior. Figure 4b shows that Java server programs also incur high instruction TLB miss rates. This phenomenon and poor instruction cache behavior together may indicate a large and scattered instruction footprint generated by JIT.

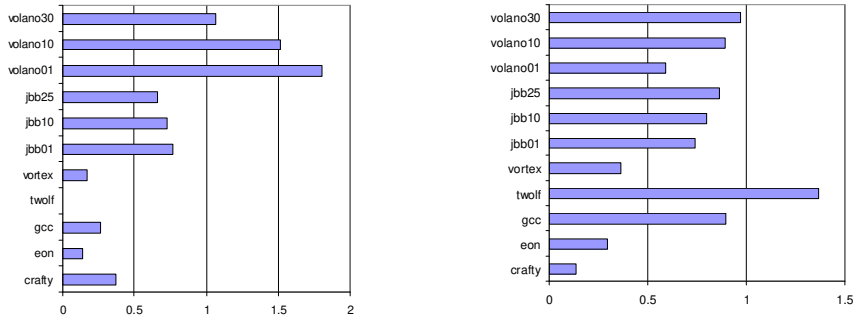


(a) Branch Behavior **(b) Speculative Factor**

Figure 5 Speculation Behavior

State-of-the-art high performance microprocessors employ speculative execution as a means to enhance performance. Though the two-level adaptive branch prediction scheme of Pentium III does a fairly good job in terms of miss prediction rate, the BTB miss rates for Java servers are

much higher compared to SPECint2000 (Figure 5a). Typical Java programs have high BTB miss rates due to difficulty in predicting indirect branches resulting from virtual function calls and code interpretation. While we do not isolate the causes of the BTB misses and mispredictions, it is clear that current BTB architectures do not work very satisfactorily for Java server code. We also investigate the extent of speculation in these applications. As in [3], we determine the speculative factor, computed as the number of instructions decoded divided by the number of instructions retired. Applications with higher branch miss prediction rate generally show higher speculative factor. But speculative factor is also sensitive to branch frequency in the code. We investigate the speculative factor of these applications in an effort to understand whether there is a marked difference in the way speculative microarchitecture is utilized by Java servers and SPECint programs. But we observe that the speculative factors of Java servers are within the range of SPECint2000 indicating that their speculative behavior is similar (Figure 5b).



(a) I-stream Stall Cycles per Instruction **(b) Resource Stall Cycles per Instruction**

Figure 6 Stall Cycles per Instruction

Figure 6 shows the I-stream stalls and resource stalls, measured in terms of the cycles in which the stall conditions occur. I-stream stalls are caused mainly by I-cache misses and ITLB misses. Resource stalls show the number of cycles in which resources such as reorder buffer entries, memory buffer entries, or execution units are not available [3]. Because of the high instruction cache miss rate and ITLB miss rate, Java server benchmarks demonstrate much worse I-stream stalls than SPECint2000 benchmarks.

In summary, Java server benchmarks show worse instruction stream behavior than SPECint2000. Higher instruction cache miss rate, higher ITLB miss rate, higher BTB miss rate and consequently, higher I-stream stall cycles are observed in multithreaded Java server applications in comparison to integer programs in the SPEC suite.

5. Impact of Multithreading on Microarchitecture

To study the impact of multithreading on processor microarchitecture, we vary the number of threads in the benchmark and study cache performance, branch predictor performance, etc. We vary the number of threads in VolanoMark and SPECjbb2000 from a small number to the maximal number that our system can properly support, and measure user and OS activity. In SPECjbb2000 less than 1% of the cycles are in OS mode. Though OS part may show different behavior, its effect on overall results is negligible in SPECjbb2000. So we only present the combined effect of OS and user code in the case of SPECjbb2000. In VolanoMark, however, the OS part constitutes more than half of all the cycles. So the OS part, user part and the total effect are shown for VolanoMark.

Number of connec-tions	CPI			Data Reference per Instruction			L1 Data Cache Misses per Data Reference			L2 Cache Miss Rate (Local)		
	User	OS	Overall	User	OS	Overall	User	OS	Overall	User	OS	Overall
20	2.84	4.68	3.68	0.629	0.833	0.722	0.047	0.059	0.053	0.049	0.039	0.043
200	3.29	4.10	3.72	0.620	0.715	0.670	0.046	0.106	0.080	0.115	0.084	0.095
400	2.79	3.56	3.23	0.615	0.651	0.636	0.042	0.146	0.104	0.128	0.078	0.093
600	2.70	3.24	3.03	0.618	0.621	0.620	0.039	0.171	0.120	0.134	0.070	0.086
800	2.56	2.97	2.82	0.612	0.581	0.592	0.037	0.190	0.132	0.144	0.072	0.089

(a) VolanoMark metrics for varying number of threads (connections)

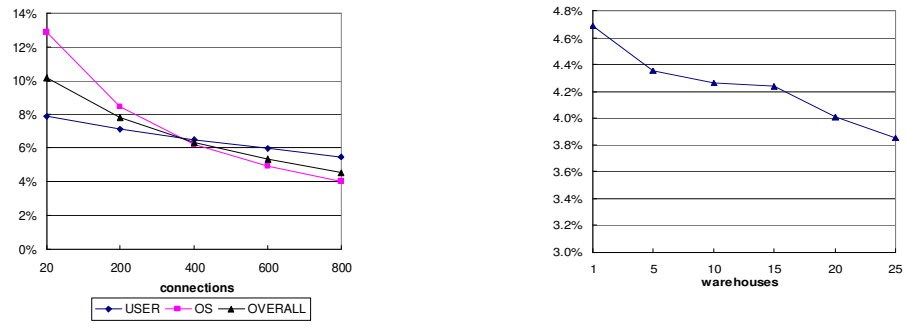
Number of Warehouses	CPI	Data Reference per Instruction	L1 Data Cache Misses per Data Reference	L2 Cache Miss Rate (Local)
1	2.28	0.5926	0.03976	0.1440
5	2.29	0.5787	0.04075	0.1591
10	2.29	0.5778	0.04094	0.1612
15	2.27	0.5723	0.04095	0.1623
20	2.28	0.5696	0.04129	0.1742
25	2.31	0.5715	0.04160	0.1888

(b) SPECjbb2000 metrics for varying number of threads or warehouses

Table 2 Impact of Multithreading

Table 2 illustrates the change in CPI and data cache performance with increasing number of threads. CPI is the compound effect of many factors including branch prediction accuracy and cache misses. Data footprint is expected to increase with increasing number of threads, hence the increase of data cache misses. But as shown in Table 2, CPI does not increase steadily. In VolanoMark we even observe a dropping OS CPI. This indicates that some of the stalls disappear with heavier multithreading.

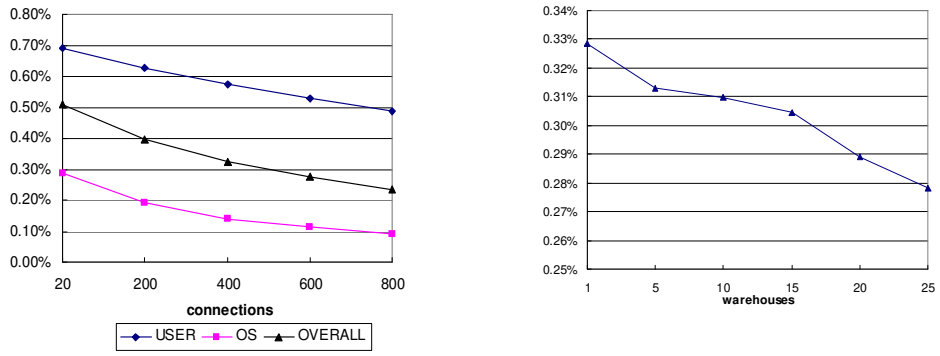
Even though the number of threads varies, the code in the program remains the same and so do the properties of the user code. Therefore, little change is observed in user code data reference per instruction (Table 2). The data set of one warehouse (25M bytes) in SPECjbb2000 is big enough to overwhelm the L1 and L2 cache. Therefore L1 data cache miss rate and L2 cache miss rate of SPECjbb2000 do not increase as quickly as those of VolanoMark when the number of threads increases. Though VolanoMark has much smaller data set for one thread, it involves larger OS part. The properties of OS code change significantly as we can see from the drastic drop in the OS data reference per instruction. The changing OS behavior and its interaction with the user part make VolanoMark's overall L1 data cache and L2 cache miss rate not very straightforward to explain. But in general, if there are more threads, the data set is correspondingly bigger and the data cache miss rates are higher also (see also Figure 13).



(a) VolanoMark **(b) SPECjbb2000**

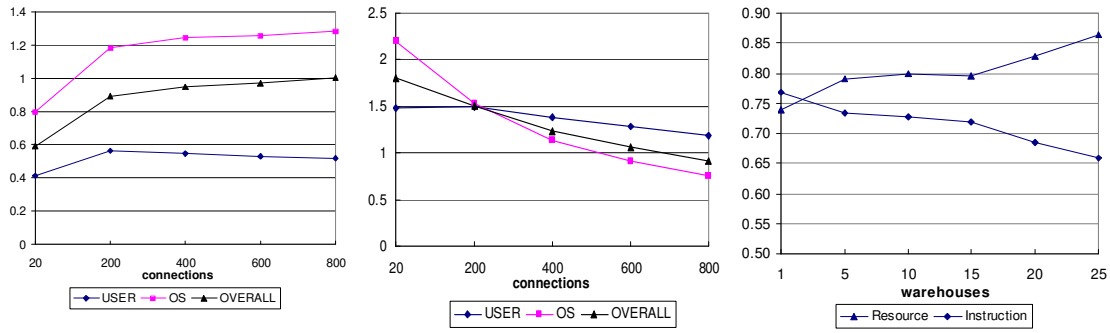
Figure 7 L1 Instruction Cache Misses per Instruction

The impact of multithreading on instruction cache behavior is very different compared to data cache behavior. Multithreaded Java servers respond to each client connection request with one or more threads. Except for a few specialized threads (e.g. logging), all threads share the same user code. This will create beneficial interference for instruction cache in that the user instructions of the currently running thread may have been fetched into L1 instruction cache by previously running threads. The more threads we have, the more likely we context switch to another thread that is executing the same part of the program thus instructions in instruction cache and entries in ITLB are reused (Figure 7,8). Since the properties of OS code change significantly, code sharing can only partly be responsible for the drastic improvement in instruction and ITLB performance in OS part in VolanoMark. Some hotspots that are more frequently executed under heavier multithreading may also have contributed to the improvement.



(a) VolanoMark **(b) SPECjbb2000**

Figure 8 Instruction TLB Misses per Instruction



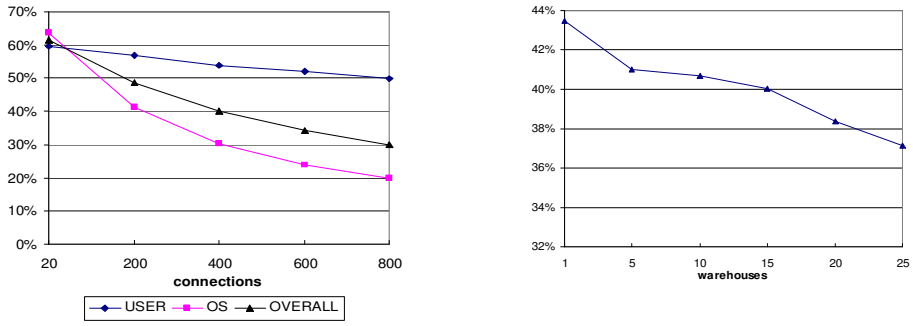
(a) VolanoMark Resource Stalls per Instruction

(b) VolanoMark Instruction Stream Stalls per instruction

(c) SPECjbb2000 Resource and Instruction Stream Stalls per Instruction

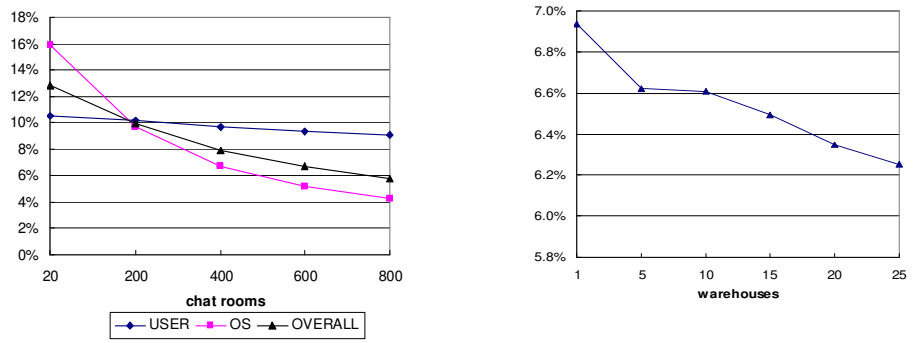
Figure 9 Resource and Instruction Stream Stalls per Instruction

As the result of beneficial interference, I-stream stalls are lowered for both OS part and user part and thus more instructions can enter execution stage in one cycle, causing more pressure on processor resources. As we can see, the resource stalls increase and soon exceed the I-stream stalls (Figure 9). The same beneficial interference also decreases BTB miss rates. And this in turn contributes to the improved branch prediction accuracy with increasing number of threads (Figure 10,11).



(a) VolanoMark (b) SPECjbb2000

Figure 10 BTB Miss Rate

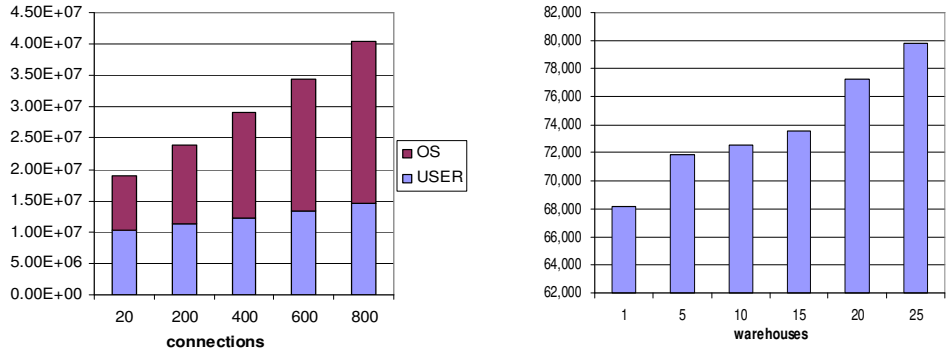


(a) VolanoMark (b) SPECjbb2000

Figure 11 Branch Miss Prediction Rate

All the above microarchitecture metrics affect throughput of the server, but instruction count variation with increased number of threads also needs to be studied. Therefore, we present the results from a different angle, normalizing all the metrics to a unit of useful work the servers perform. In VolanoMark measurement, each simulated client user sends the same number of messages. Thus the server performs the same amount of useful work for each connection. In

SPECjbb2000, there are five types of transactions. To measure the average effect, we do not distinguish each transaction but normalize the results by the total number of transactions.



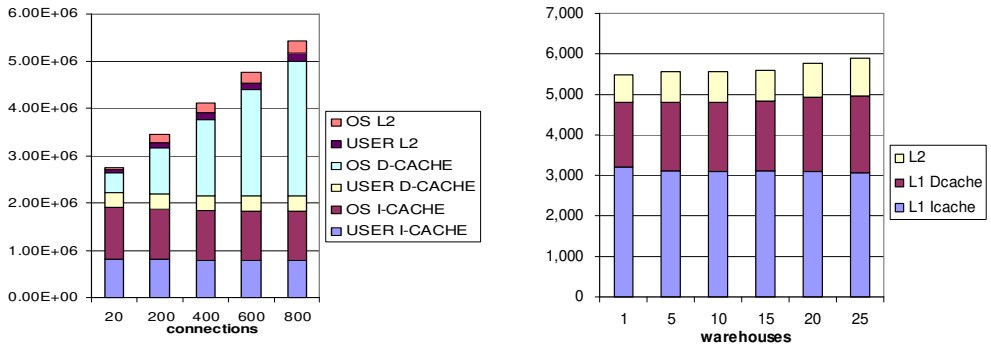
(a) VolanoMark: Instructions per Connection **(b) SPECjbb2000: Instructions per Transaction**

Figure 12 Instructions per Unit Work

The number of instructions executed for each unit of useful work increases in both cases. The most significant increase is in the OS part in VolanoMark (Figure 12). The increase may be due to network communications, thread scheduling and thread synchronization. With increased number of simultaneous clients, the servers have to spend more time in this overhead. This phenomenon will certainly affect the scalability negatively. Operating system code can probably be optimized to alleviate the problem.

Cache misses per unit work also change with increased number of threads. When the number of threads is small, instruction cache misses constitute the major part of misses. As the number of threads increase, the fraction of L1 data cache misses and L2 cache misses increases while the fraction of instruction cache misses drops slowly (Figure 13). In VolanoMark the OS L1 data cache misses increase so fast that it soon replaces instruction cache misses as the dominant component in the overall L1 cache misses. A possible reason is that the operating system needs to manage a large number of network messages and may need to copy them between user and OS space. Please note that the increase in number of data and L2 cache misses are accompanied

by the increase in number of instructions (and data references). This explains why no dramatic cache miss rate increase is observed in Table 2.



(a) VolanoMark: Cache Misses per Connection **(b) SPECjbb2000: Cache Misses per Transaction**

Figure 13 Cache Misses per Unit Work

6. Conclusions

Multithreading is an important characteristic of server applications, especially for Java servers due to the lack of effective mechanisms in Java to share threads among client connections. In this paper, we present the detailed workload characterization of two multithreaded Java server benchmarks: VolanoMark and SPECjbb2000. We measured their transaction activity on a state-of-the-art HotSpot JVM on a real superscalar uniprocessor machine. We compared Java servers to SPECint2000 and studied the impact on processor with increasing number of threads. Our main findings are as follows:

1. Java server benchmarks show worse instruction stream behavior than SPECint2000. Higher instruction cache miss rate, higher ITLB miss rate, higher BTB miss rate and consequently, higher I-stream stall cycles are observed in multithreaded Java server applications in comparison to integer programs in the SPEC suite.
2. With increasing number of threads, the instruction behavior improves due to code sharing and constructive interference. All the aforementioned miss rates and stalls drop. If the server has network connections as in VolanoMark, the phenomenon is stronger.

3. Resource stalls in the processor increase with increased number of threads, and eventually exceed the diminishing I-stream stalls.
4. When there are more connections/clients, the number of instructions that the processor executes per client/thread increases, which affects the scalability of the system negatively. Enhancement in OS may be required to alleviate this problem.

Acknowledgement

This work is partially sponsored by Tivoli Corporation, IBM, Intel, Microsoft, Sun Microsystems, and by National Science Foundation under grant numbers CCR-9796098 and EIA-9807112.

References

[1] A. Aliamaki, D. J. DeWitt, M. D. Hill and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th VLDB Conference*, Edinburgh, Scotland, 1999.

[2] L. A. Barroso, K. Gharachorloo and E. Bugnion, Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[3] D. Bhandarkar and J. Ding. Performance Characterization of the Pentium Pro Processor. In *Proceedings of The third International Symposium on High-Performance Computer Architecture*, 1997.

[4] H. W. Cain, R. Rajwar, M. Marden, M. H. Lipasti. An Architectural Evaluation of Java TPC-W. In *Proceedings of The Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[5] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, Y. Won. Detailed Characterization of a Quad Pentium Pro Server Running TPC-D. In *Proceedings of International Conference on Computer Design*, 1999

[6] S. Hily, A. Seznec. Branch Prediction and Simultaneous Multithreading. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, 1996

[7] Intel Corporation. Intel Architecture Software Developer's Manual (Volum3: System Programming), 1999

[8] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[9] T. Li, L. K. John, N.Vijaykrishnan, A. Sivasubramaniam, A.Murthy, and J. Sabarinathan, Using Complete System Simulation to Characterize SPECjvm98 Benchmarks. In *Proceedings of International Conference on Supercomputing*, 2000,

[10] A. M. G. Maynard, C. M. Donnelly and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, October 1994.

[11] S. Oaks and H. Wong. Java Threads, 2nd Edition, O'Reilly & Associates, January 1999

[12] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural Issues in Java Runtime Systems. In *Proceedings of the Sixth International Conference on High Performance Computer Architecture*, January 2000.

[14] P. Ranganathan, K. Gharachorloo, S. V. Adve and L. A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[15] PMON webpage. http://www.ece.utexas.edu/projects/ece/lca/pmon

[16] B. Rychlik and J. P. Shen. Characterization of Value Locality in Java Programs, Workshop on Workload Characterization, ICCD, September 2000.

[17] Sun Microsystems, Inc. JDK1.4.0 Beta Release Notes. 2001

[18] Volano LLC. VolanoMark benchmark. http://www.volano.com/benchmarks.html

[19] SPEC Benchmarks, http://www.spec.org