Workload Characterization of Multithreaded Java Servers

Yue Luo and Lizy Kurian John
Department of Electrical and Computer Engineering
The University of Texas at Austin
{luo,ljohn}@ece.utexas.edu

Abstract

Java has gained popularity in the commercial server arena, but the characteristics of Java server applications are not well understood. In this research, we characterize the behavior of two Java server benchmarks, VolanoMark and SPECjbb2000, on a Pentium III system with the latest Java Hotspot Server VM. We compare Java server applications with SPECint2000 and also investigate the impact of multithreading by increasing the number of clients. Java servers are seen to exhibit 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 increased locality of access. But the resource stalls increase and eventually dwarf the diminishing I-stream stalls. With more clients, the instruction count per unit work increases and becomes a hindrance to the scalability of the servers.

1. Introduction

Java has emerged as a competitive paradigm for server platforms because of its “write once run everywhere” property and its enhanced security features. Typically, Java execution suffers from software translation overhead, but the overhead of Just-In-Time (JIT) compilation can be easily amortized in long-running server applications. Moreover, Java provides the 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 the 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 the 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 the performance under the condition of a large number of threads is crucial for a Java server to support multiple clients simultaneously. Sun Microsystems Inc. is proposing a solution for this problem in JDK 1.4 beta [16].

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 applications on the processor performance. For this purpose, we perform two types of experiments. First we compare the 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 the next section we introduce some related work. Section 3 provides our experimental methodology as well as some background on the benchmarks we use. Sections 4 and 5 present our main results for the two types of experiments. Section 6 summarizes the key observations and offers the 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 [1, 2, 5, 8, 10, 13]. The 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 the subject of active research for years. Most of the results of Java workload characterization are based on the study of SPECjvm98 [9, 12, 15], 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. 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 the client level, software translation is seen to dominate when the applications are short-running [12]. However, we do not expect the same dominance at the server level.

Commercial pure Java servers are just emerging and so are the research efforts in quantifying their behavior, especially the multithreading impact. Cain et al. [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. However, some constructive interference between threads in multithreading was reported in the past by Hily et al. [6]. They 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.

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 a 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. To compare the two benchmarks to more "traditional" applications, we also measured SPECint2000 on the same platform. SPECint2000 comprises such a wide variety of applications that we did not expect the Java server applications to stand out in terms of microarchitecture metrics. We do find, however, that these Java servers have some unique properties.

VolanoMark [17] is a pure Java server benchmark with long-lasting network connections and high thread counts. It can be divided into two parts, a server and a 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 the server and waits for the server to broadcast the messages to all the users in the same chat room. The VolanoMark server creates two threads for each client connection. VolanoMark can be run to test both the speed and the scalability of a system. In the speed test, it is run in a loopback fashion with the server and the client on the same machine. In the scalability test, on which our experiment is based, the server and the client are run on two separate machines with high-speed network connections.

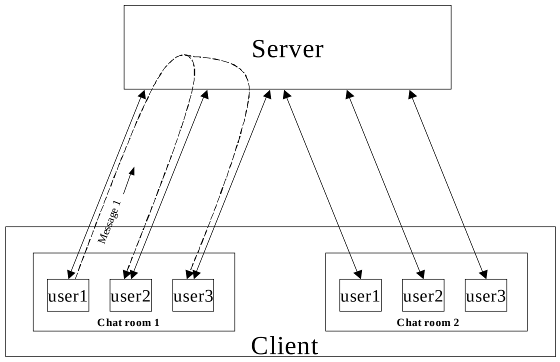


Figure 1. VolanoMark environment

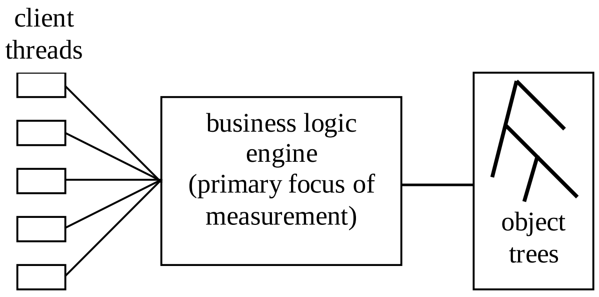


Figure 2. SPECjbb2000 environment [19]

SPECjbb2000 (Java Business Benchmark) [18] 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 25MB data set stored in binary trees (Btrees). 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.

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 32-byte block size. The operating system on the system under measurement is Windows NT Workstation 4.0 with Service Pack 6a. We use the Sun JDK 1.3.0 with Hotspot Server (build 2.0fcs-E, mixed mode) as the Java virtual machine.

3.3. Monitoring method

The Intel P6 family processor has 2 performance counters. Events in nonprivileged user code (user mode) and privileged operating system code (OS mode) can be counted separately. Our lab developed PMON [14] to access these counters. PMON consists of two parts, a device driver and a control program. The driver reads the performance counters [3,7] of the 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 memory swapping is 25. Therefore, we increase the number of warehouses from 1 to 25 in our experiments.

Though it is desirable to have quick starting and shutdown processes, the most important aspect of server performance is how the server responds to client requests. To synchronize our measurements with the client connections in the 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, the counters are sampled every 3 seconds and these samples are accumulated to get the final results. 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 we skipped the starting of the program, where most compilation is done, and if any compilation slipped into our measurement, it would only account for a very small part in the long running of the benchmarks.

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].

Table 1 shows the percentage of cycles spent in OS mode. SPECjbb2000 has neither file accesses nor network connections. And since it is a memory resident

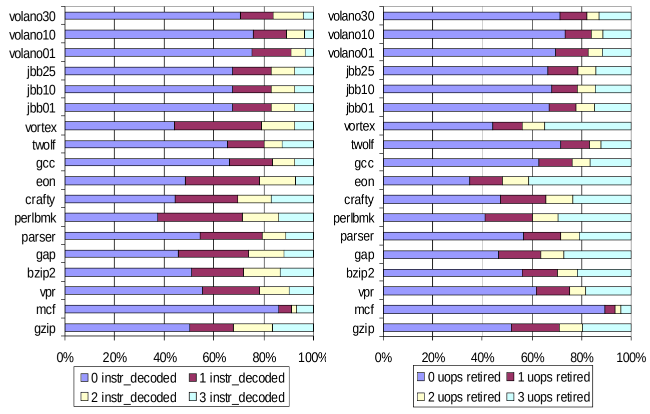
Java database program, few page faults occur. Therefore, OS cycle time constitutes less than 0.7% of the total execution time, which is not very different from SPECint2000. VolanoMark, on the other hand, has significant OS part because it spends most of its time in receiving and sending network messages, 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 OS mode.

Table 1. Java servers vs. SPECint2000

Benchmark		OS Cycle Percent	CPI	Data References per Instruction	Memory Transactions per 1000 Instructions
Java Servers	volano30	65.32%	3.03	0.620	10.67
	volano10	58.25%	3.72	0.670	18.39
	volano01	57.86%	3.68	0.722	9.51
	jbb25	0.67%	2.31	0.572	15.97
	jbb10	0.63%	2.29	0.578	14.14
	jbb01	0.57%	2.28	0.593	13.14
SPECint 2000	vortex	1.06%	1.27	0.674	2.80
	twolf	0.37%	2.25	0.593	17.22
	gcc	1.04%	2.25	0.629	11.36
	eon	0.24%	1.36	0.839	0.01
	crafty	0.27%	1.22	0.499	0.35
	perlbmk	0.81%	1.13	0.535	1.06
	parser	0.41%	1.64	0.638	6.61
	gap	0.47%	1.32	0.646	9.48
	bzip2	0.44%	1.36	0.579	7.28
	vpr	0.38%	1.79	0.697	10.18
	mcf	0.34%	6.65	0.620	95.92
	gzip	0.48%	1.24	0.382	1.21

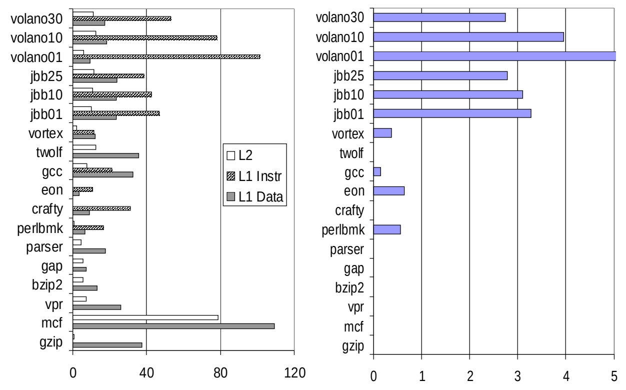
In table 1 Java server benchmarks show higher CPI than most of SPECint2000 programs. Operating system code on servers is known to have worse CPI than user code [8]. With a much larger OS part, VolanoMark shows much higher CPI than SPECjbb2000.

The Pentium III processor has a microarchitecture similar to the Pentium Pro discussed in [3]. 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. But in Java servers, instruction level parallelism is seen to be 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 mcf in SPECint2000 shows worse decode/retirement profile, but, as we will see in Figure 4, it is caused by the extraordinarily high L1 data cache misses and L2 cache misses.



(a) Instruction decode profile (b) Instruction retirement profile

Figure 3. Instruction decode/retirement behavior

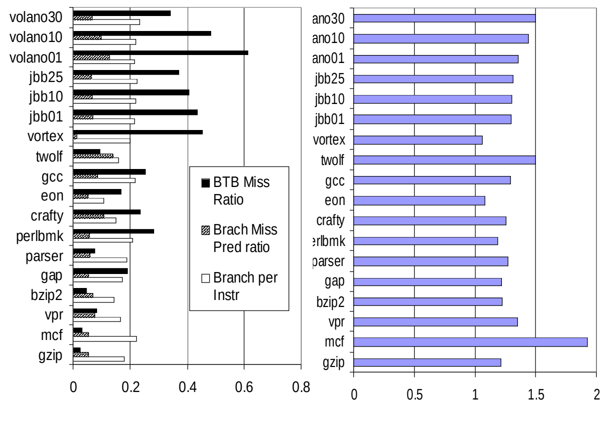


(a) Cache misses per 1000 instructions (b) ITLB misses per 1000 instructions

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. The L1 data cache and L2 cache misses per instruction of the Java servers are within the range of SPECint2000 although on the higher end (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 about the Java servers is that they 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 [12]. 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 suggest a large and scattered instruction footprint generated by JIT.

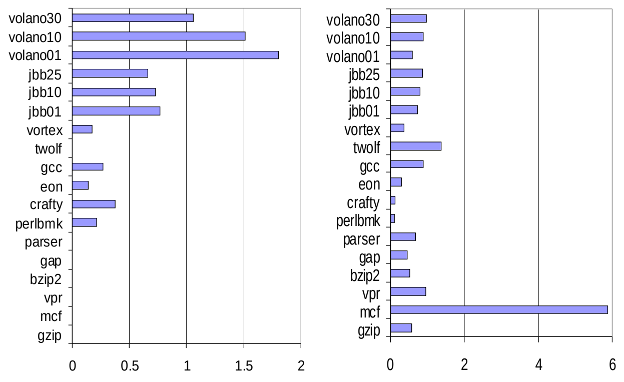
State-of-the-art high performance microprocessors employ speculative execution as a means to enhance performance. Though the two-level adaptive branch prediction scheme of Pentium III does a fairly good job in terms of miss prediction rate, the BTB miss rates for Java servers are higher compared to SPECint2000 (Figure 5a). While we do not isolate the causes of the BTB misses and mispredictions, it is clear that the current BTB architecture does 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 the Java servers and the SPECint programs. 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) Branch behavior (b) Speculative factor

Figure 5. Speculation behavior

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 instruction 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 higher I-stream stalls than SPECint2000 benchmarks.



(a) I-stream stall cycles per instruction (b) Resource stall cycles per instruction

Figure 6. Stall cycles per instruction

In summary, the 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 the multithreaded Java server applications in comparison to the integer programs in the SPECint2000 suite.

5. Impact of multithreading on the processor microarchitecture

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

Table 2 illustrates the changes 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 in 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.

Table 2. Impact of multithreading

(a) CPI and data references per instruction for VolanoMark

Number of connections	CPI			Data References per Instruction		
	User	OS	Overall	User	OS	Overall
20	2.84	4.68	3.68	0.629	0.833	0.722
200	3.29	4.10	3.72	0.620	0.715	0.670
400	2.79	3.56	3.23	0.615	0.651	0.636
600	2.70	3.24	3.03	0.618	0.621	0.620
800	2.56	2.97	2.82	0.612	0.581	0.592

(b) L1 data cache misses per data reference and L2 cache miss rate for VolanoMark

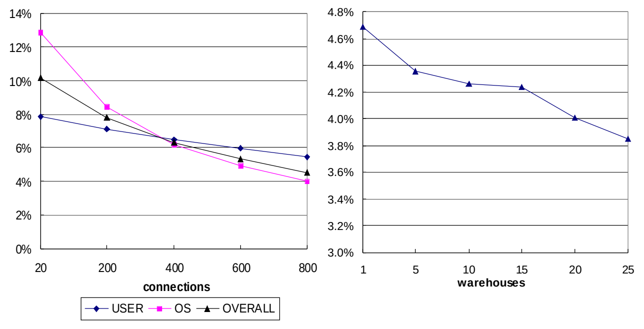
Number of connections	L1 Data Cache Misses per Data Reference			L2 Cache Miss Rate (Local)		
	User	OS	Overall	OS	User	Overall
20	0.047	0.059	0.053	0.049	0.039	0.043
200	0.046	0.106	0.080	0.115	0.084	0.095
400	0.042	0.146	0.104	0.128	0.078	0.093
600	0.039	0.171	0.120	0.134	0.070	0.086
800	0.037	0.190	0.132	0.144	0.072	0.089

(c) Metrics for SPECjbb2000

Number of Warehouses	CPI	Data References 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

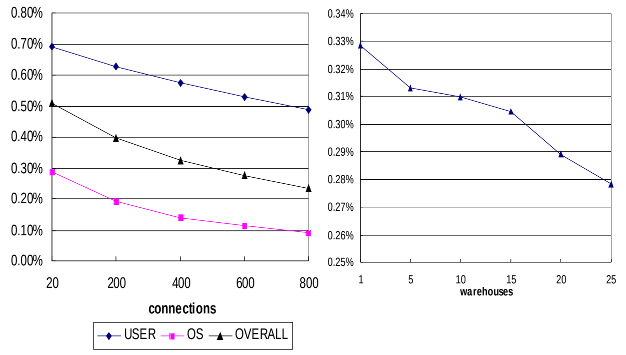
The data set of one warehouse (25M bytes) in SPECjbb2000 is big enough to overwhelm the L1 and L2 cache. Therefore L1 data cache miss rate and L2 cache miss rate of SPECjbb2000 do not increase as quickly as those of VolanoMark when the number of threads increases. Though VolanoMark has much smaller data 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 also higher (see also Figure 13).

As shown in Figure 7, the impact of multithreading on the instruction cache behavior is very different compared to the 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 some 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. With more client connections, some relatively small parts in the programs get executed more frequently. For example, when there are more warehouses in SPECjbb2000, the underlying Btrees become bigger and thus more time is spent in the Java methods accessing these Btree structures. This localized execution also contributes to the improvement in the instruction cache behavior. For the OS part in VolanoMark, the hotspot effect is more phenomenal, indicated not only by the sharp drop in the instruction cache miss rate but also by the changes in the number of data references per instruction. With VTune we find that at 600 client connections more than 35% of the OS cycles are spent in afd.sys, a small driver (66KB) providing sockets emulation. While at 20 client connections, afd.sys accounts for only 11% of the OS cycles. Accompanying the improvement in the instruction cache behavior is the decrease in the number of instruction TLB misses per instruction (Figure 8).



(a) VolanoMark (b) SPECjbb2000

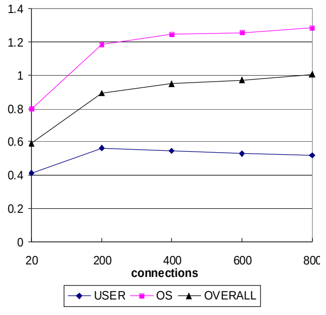
Figure 7. L1 instruction cache misses per instruction



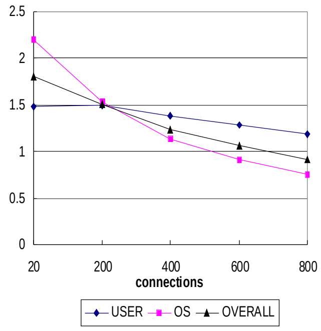
(a) VolanoMark (b) SPECjbb2000

Figure 8. Instruction TLB misses per instruction

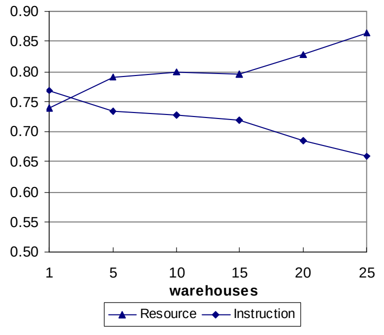
As a result, the I-stream stalls are lowered for both the OS part and the user part and thus more instructions can enter the execution stage in one clock cycle, causing more pressure on processor resources. As we can see, the resource stalls increase and soon exceed the I-stream stalls (Figure 9). The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



(a) VolanoMark resource stalls per instruction

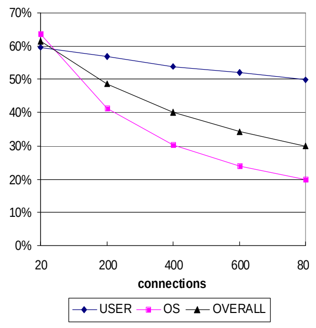


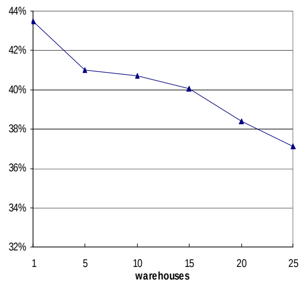
(b) VolanoMark instruction stream stalls per instruction



(c) SPECjbb2000 resource and instruction stream stalls per instruction

Figure 9. Resource and instruction stream stalls per instruction

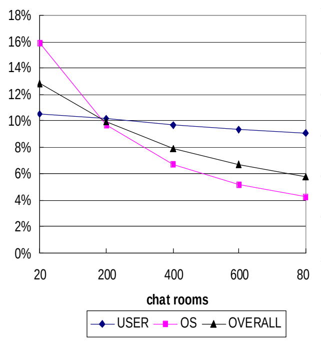


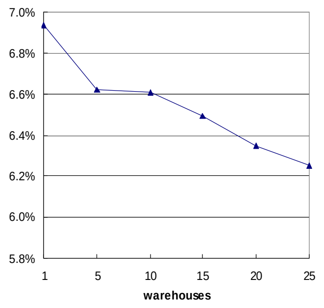


(a) VolanoMark (b) SPECjbb2000

Figure 10. BTB miss rate

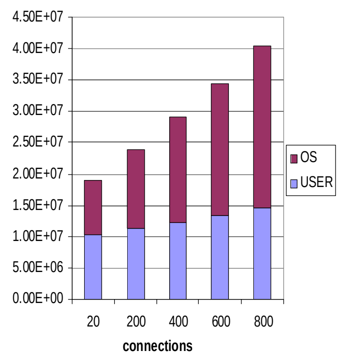
All the above microarchitecture metrics affect throughput of the server, but instruction count variation with increased number of threads also needs to be studied. Therefore, we present the results from a different angle, normalizing all the metrics to a unit of useful work the servers perform. In VolanoMark 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.

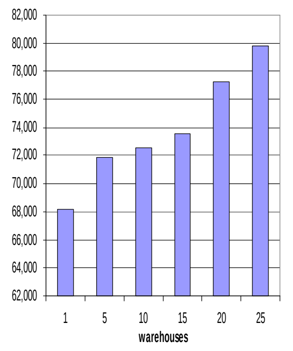




(a) VolanoMark (b) SPECjbb2000

Figure 11. Branch miss prediction rate



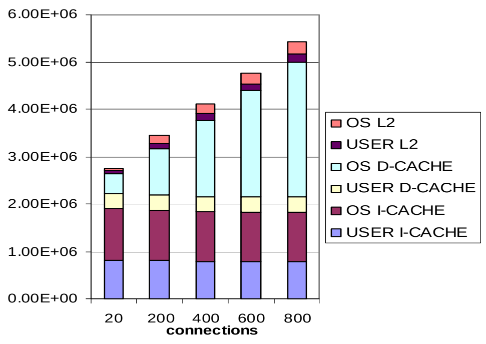


(a) VolanoMark (b) SPECjbb2000

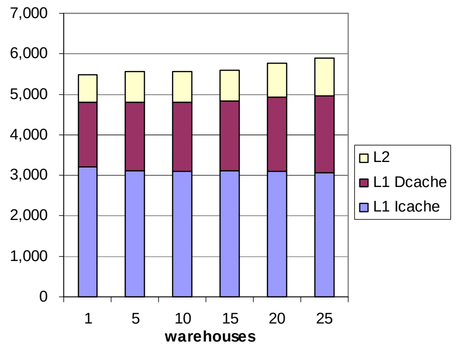
Figure 12. Instructions per unit work

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). This increase is largely due to network communications, thread scheduling and synchronization. With increased number of simultaneous clients, the VolanoMark server has to spend more time in this overhead. For SPECjbb2000, in addition to a higher thread management overhead, the bigger Btree structures require more instructions for each search and update operation. In either case, the instruction count increase will certainly affect the scalability of the Java server negatively. Operating system code as well as the Java virtual machine may be further 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 cache 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 the number of L1 data cache and L2 cache misses is accompanied by the increase in the number of instructions (and data references). This explains why no dramatic cache miss rate increase is observed in Table 2.



(a) VolanoMark



(b) SPECjbb2000

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 the current 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 the multithreaded Java server applications in comparison to the integer programs in the SPECint suite.
2. With increasing number of threads, the instruction behavior improves due to increased locality of access. 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 unit of work (e.g. one transaction) increases, which affects the scalability of the system negatively. Further enhancement in the operating system and the Java virtual machine may be expected 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. Chapter 1 in 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.

[13] 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.

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

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

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

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

[18] Standard Performance Evaluation Corporation. SPECjbb2000 Benchmark. http://www.spec.org/osg/jbb2000/

[19] Standard Performance Evaluation Corporation. Architecture schematic of the SPEC JBB2000 benchmark process. http://www.spec.org/osg/jbb2000/images/arch.jpg.