Performance Characterization of Java Applications on SMT Processors

Wei Huang, Jiang Lin, Zhao Zhang, and J. Morris Chang
Department of Electrical and Computer Engineering
Iowa State University
Ames, Iowa 50011
{huangwei, linj, zzhang, morris}@iastate.edu

Abstract

As Java is emerging as one of the major programming languages in software development, studying how Java applications behave on recent SMT processors is of great interest. This paper characterizes the performance of Java applications on an Intel Pentium 4 Hyper-Threading processor. Using the performance counters provided by Pentium 4, we quantitatively evaluate micro-architecture metrics while running various types of Java applications. The experimental results reveal that: (1) Hyper-Threading can indeed improve the performance of multithreaded Java programs; (2) The resource contentions within Pentium 4 are the major reason of pipeline inefficiency, which prevents better performance promised by SMT; (3) The static partition design of Hyper-Threading causes considerable performance loss for many single-thread Java programs; (4) Most multiprogrammed Java benchmarks can achieve decent combined speedups on Hyper-Threading processors.

1. Introduction

In recent years, *simultaneous multithreading* (SMT) has become popular in modern processor designs [28]. The motivation of SMT is based on the observation of low processor resource utilization in traditional superscalar processors. By allowing multiple threads to issue instructions simultaneously to keep pipeline busy, SMT can hide the memory hierarchy latency, reduce branch mis-predication penalty, and better utilize processor resources. Compared with conventional superscalar designs, this technology is expected to deliver higher performance at the cost of a small amount of extra transistors. Because of these benefits, SMT has been adopted in the current generation of microprocessors, including Intel's Pentium 4 and IBM's Power5.

Java is emerging as one of the major programming languages for software development. Designed with many advanced features, such as automatic memory management, cross-platform portability and enforced security check, Java has become a popular programming language used on various platforms. While many studies show that SMT can substantially increase the performance of various types of applications, how Java applications perform on SMT platform has not been well investigated. This topic is of interest for three reasons. First, Java provides built-in multithreading support for programming, which means programmers can develop multithreaded applications at language level. SMT is especially suitable for Java applications that are inherently multithreaded. Secondly, in addition to normal Java application threads, many helper threads exist inside the *Java Virtual Machine* (JVM). For instance, the JVM has a *garbage collection* (GC) thread responsible for recycling the un-referenced heap space. That means the whole JVM usually is a multithreaded application even when the Java applications on the top of it are single-threaded. Thirdly, because many components of the JVM are involved in executing Java bytecodes, Java programs can behave differently from traditional C/C++ applications. Such complexity of execution requires further understanding of how Java programs behave on different architectures, including SMT.

In this paper, we examine the characterization of Java applications running on SMT-capable processors. By turning on and off the Hyper-Threading (HT) support, we study the processor behaviors using the hardware performance counters provided by Intel Pentium 4 processors. We mainly focus on multithreaded and multi-programmed Java programs. Additionally, the impact of resource contention within SMT processors on single-threaded Java programs is also investigated. The observations presented in this paper can be utilized while designing new VM, OS, or processor architecture. For instance, we find that most single-threaded Java programs perform worse when Hyper-Threading is turned on. This fact indicates that virtual machine should be further optimized for such types of applications. Similarly, the poor L1 cache performance while running multithreaded Java programs suggests that incorporating larger L1 cache may be effective to alleviate memory latency.

The rest of this paper is organized as follows. Section 2 briefly reviews related research work. Section 3 describes the experimental methodology and benchmarks used in this study. The experimental results are presented in Section 4. Finally, Section 5 summarizes key observations and concludes this paper.

2. Related work

The behaviors of Java applications have been well evaluated since Java was first introduced in late 1995 [14][21][19][8]. Those studies focused on singled threaded Java programs, especially the SPECjvm98 benchmarks [25]. In recent years, because of the popularity of Java-based server applications, the performance of multithreaded Java programs are becoming of great interest.

Using the performance counters provided by the processor, Luo *et al.* studied the characterization of Java server applications on the Pentium III [16]. They found that such programs have worse instruction streams (including I-Cache miss rate and ITLB miss rate, etc.) than SPECint2000. By increasing the number of threads, they also studied the impact of Java threads on the micro-architecture. The experimental results showed that increasing the number of threads will introduce more resource contentions even though the instruction stream performance can be improved due to the increased access locality.

A similar work by Karlsson *et al.* studied the memory system behaviors of two Java middleware (SPECjbb2000 and ECperf) running on SMP systems [13]. They observed poor scalabilities for both benchmarks when the number of processors is greater than 12. The experimental results showed that resource contention and memory stall time are two major factors affecting system scalability. Additionally, they found that cache to cache transfer rate is pretty high for Java application servers.

The workload characterization of non-Java applications on SMT processors have been evaluated by many researchers [15][20][6]. Those studies mainly focused on the interactions between software and SMT processors, as well as the potential performance bottlenecks. Redstone *et al.* identified that OS constitutes a large portion of server applications execution time [20]. Due to the very large instruction and data memory footprint, the OS has worse cache and TLB performance than SPECInt workload. However, they also found that latency tolerance, which is caused by cooperative sharing in SMT processors, has positive impacts on performance. Overall, Apache server has over 4-fold throughput improvement on an 8-way issue, 8-context SMT processor.

Jack Lo *et al.* studied the database performance for OLTP and DSS workloads [15]. Similar to [20], they found that an SMT processor can increase the IPCs of OLTP and DSS by 3-fold and 1.5-fold, respectively. However, SMT introduces severe cache conflicts using conventional virtual memory management. They proposed a new page placement policy which can reduce the cache misses significantly.

Tuck *et al.* recently examined the performance of Intel Pentium 4 processors using SPEC2000 [27]. They used various mixes of workloads (multiprogrammed, multithreaded and parallel) to validate and evaluate the effectiveness of Hyper-Threading design. Their results showed that if the issue width and the number of hardware contexts are taken into consideration, the measured performance on Pentium 4 matches the simulation-based expectations. While a static partition is important to minimize conflicts and control throughput losses, they believed that a dynamic partition might be required for an 8-context configuration. A closely related study can be found in [3].

To our knowledge, no previous work has evaluated the characterization of Java applications running on SMT processors. Our objective is to understand the behaviors of Pentium 4 when it is executing Java programs. Other issues, including the impacts of software configurations and job pairing, are also conducted in this study.

3. Experimental environment

3.1. Java benchmarks

We use ten Java benchmarks in our experiments. Table 1 gives the descriptions of these benchmarks, including their input parameters. According to their properties, these benchmarks can be divided into two groups: single-threaded and multithreaded. Six programs of SPECjvm98 are incorporated as single-threaded benchmarks. These programs are representative for measuring the performance of JVM client platforms.

Benchmark	Description	Input	
compress	Java port of the SPEC95 compress program using modified LZW method	-s100 -m1 -M1	Single-threaded
jess	A Java expert shell system based on NASA's CLIPS expert system	-s100 -m1 -M1	
db	Performs a series of functions on a small database	-s100 -m1 -M1	
javac	The Java compiler from the JDK 1.0.2	-s100 -m1 -M1	
mpegaudio	An ISO MPEG Layer-3 audio decoder	-s100 -m1 -M1	
jack	A Java parser generator that is based on earlier version of JavaCC	-s100 -m1 -M1	
MolDyn	An N-body program modeling particles interacting under a Lennard-Jones potential	N = 2,048	Multi-
MonteCarlo	A product price deriving program based on Monte Carlo techniques	N = 10,000	
RayTracer	A 3D raytracer, which renders 64 spheres with configurable resolutions	N = 150	
PseudoJBB	A variant of SPECjbb2000 with fixed size of working set	100,000 trans.	

Of the four multithreaded benchmarks we use, three (MolDyn, MonteCarlo, and RayTracer) come from multithreaded Java Grande Forum (JGF) Benchmark Suite [10]. We also use PseudoJBB, a variant of SPECjbb2000 [24], as a multithreaded benchmark. Different from SPECjbb2000, PseudoJBB can run a fixed number of transactions in multiple warehouses. Therefore, the execution time of PseudoJBB can be used for comparison purposes. This approach has been adopted by other researchers in recently published papers ([2][26][7]). We explicitly exclude the data initialization part of PseudoJBB benchmark to study the real performance of long-running server applications. Notice that these multithreaded applications allow users to specify the number of threads as a parameter. That means they can be used as single-threaded benchmarks when the number of threads is 1. We will use them in this way in later experiments.

3.2. Experimental platform

The operating system we use is RedHat Linux 9 running on a single 2.8GHz Intel Pentium 4 Hyper-Threading processor. Instead of using a traditional cache design, the Pentium 4 adopts trace cache as the level 1 instruction cache. The CPU has a 64-byte L1 and L2 cache line size, 8KB 4-way set associative L1 data cache, 12K *micro-operations* (*μop*s) L1 instruction trace cache, and a 1MB 8-way set associative unified L2 on-chip cache. The machine has 1GB of dual channel 400MHz DDR RAM and 800MHz front side bus. To reduce the impacts from other processes in OS, the machine is booted in single user mode and disconnected from network during experiments. The JVM we use is Sun *Java Runtime Environment* (JRE) 1.4.2_05 and the heap size is configured to be 512MB for all benchmarks.

3.3. Tool for performance measurement

Performance counters are becoming an indispensable component in modern processor designs. Equipped with 18 performance counters, which is much larger than 2 counters of its predecessor, the Pentium III, the Pentium 4 can monitor 48 events at the architecture level [22]. In addition, many new features are introduced in the design of performance monitoring, such as precise event-based sampling and *μop* tagging mechanism. These features can help programmers to further understand the characterization of programs. *Brink and Abyss* is a tool developed by Sprunt [23] to facilitate users' employment of the performance monitoring mechanisms of the Pentium 4. With this tool, users can monitor various aspects of Pentium 4 processors, especially the Hyper-Threading feature. Compared to the Intel VTune performance analyzer [9], Brink and Abyss provides necessary functionalities while incurring much less overhead. In the rest of this paper, unless otherwise stated, all data is obtained using this tool.

4. Experimental Results

This section examines the performance of various types of Java applications running on Intel Hyper-Threading processors. We first study the micro-architecture behaviors of multithreaded Java programs. SMT technology is double-sided: it reduces IPC losses because of improved resource utilization efficiency, but increases pressure on those resources. For instance, SMT can reduce processor stall cycles per cache miss, but may increase the number of cache misses caused by the cache contention. In the performance metrics we examine, both sides are observed while running multithreaded Java programs. We then analyze the combined speedups while running multiple independent Java programs on Hyper-Threading processors. Three benchmarks are identified as bad partners to the applications they pair with. Other issues, such as the impact of static resource partition and the number of threads, are also examined in this section.

4.1. Detailed characterization of multithreaded Java applications

We run multithreaded benchmarks with 2 and 8 threads to measure the SMT performance. Notice that the processor has two contexts, thus 8 threads are multiplexed onto the two contexts. Table 2 shows key characteristics of each benchmark running on Pentium 4 CPU with HT-enabled. The fourth column indicates the percentage of cycles each program spent in OS mode. For three benchmarks (MolDyn, MonteCarlo, and PseudoJBB), OS time constitutes less than 3% of the total execution time. RayTracer, however, has more OS activities because each of its thread maintains a copy of scene data as the temporary storage for parallelization. Another observation is that OS cycle percentage increases with the number of threads. Apparently, this is caused by more frequent thread scheduling, which is the responsibility of OS.

Table 2. Characterization of multithreaded benchmarks on Hyper-Threading processor

Benchmark	App. Thread #	CPI	OS cycle %	CPU DT mode %
MolDyn02	2	1.13	2.01%	94.85%
MolDyn08	8	1.39	2.50%	93.47%
MonteCarlo02	2	2.96	2.74%	90.56%
MonteCarlo08	8	2.09	2.92%	92.52%
RayTracer02	2	2.19	11.61%	72.92%
RayTracer08	8	2.19	11.88%	72.55%
PseudoJBB02	2	2.69	2.66%	90.29%
PseudoJBB08	8	2.77	2.98%	90.55%

The fifth column in Table 2 (*CPU DT mode percent*) quantifies the percentage of cycles the CPU is running under dual-thread mode, which means both logical processors are executing instructions. For instance, while running MolDyn02, both logical processors are active around 95% of the total execution time. This parameter indicates how well the program has been parallelized, as well as the performance improvement potentials it can gain on an SMT processor. The DT mode percents of all benchmarks, except RayTracer, are greater than 90%. RayTracer has poorer parallelism because it spends a larger percentage of execution time on collaboration and synchronization.

Figure 1 shows the *instructions per cycle* (IPC) of multithreaded benchmarks running on a Pentium 4 processor with HT-disabled and HT-enabled, respectively. The results indicate that, for multithreaded Java applications, Hyper-Threading technology does improve their performance. However, compared with the performance enhancement reported by other researchers, this improvement is relatively small. This implies that the performance characterization of Java server applications on Hyper-Threading processors require closer examinations.

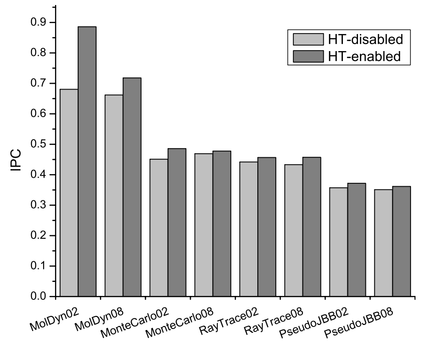

Figure 1. IPCs of multithreaded benchmarks on Pentium 4 processors

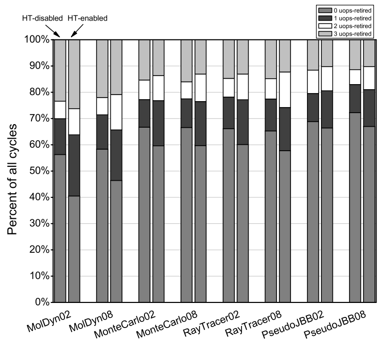

Figure 2. Instruction retirement profile

To further gain performance insight, we use Brink and Abyss to collect the retirement profile of Java benchmarks (shown in Figure 2). The Pentium 4 can retire up to 3 *μops* per clock cycle. The retirement profile is helpful for understanding the performance of processor at the last stage. According to Figure 2, the instruction level parallelism of multithreaded Java programs is limited on the Pentium 4 processor. With HT disabled, CPU does not commit any *μop* for around 60% of the total execution time. This explains why multithreaded Java benchmarks have such low IPCs. This problem is alleviated when CPU is running under Hyper-Threading mode. On average, the portions of retiring 1 and 2 *μop*s are improved by 47.53% and 50.1%, respectively. Note the portion of retiring 3 *μop*s drops by 7.52%, which indicates SMT makes the execution smoother and thus more instructions retire in 1 or 2 *μop*s groups. Thus enabling Hyper-Threading leads to an overall improvement in performance.

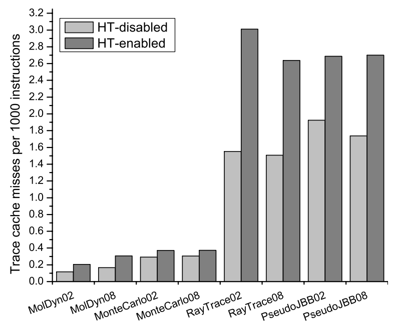

Figure 3. Trace cache misses per 1,000 instructions

In SMT processors, cache behaviors become complicated due to cache sharing. On one hand, the cache miss rate would increase because of the contentions from multiple hardware contexts. In [15], it was reported that SMT introduces severe cache conflicts for conventional virtual memory management for on-line transaction processing (OLTP) workload. On the other hand, sharing might result in constructive cache interferences, which will improve cache behaviors. For instance, if multiple threads are executing the same set of instructions, they might prefetch the instruction blocks for each other and help tolerate long memory latency. Interestingly, both patterns are observed on L1 and L2 caches of the Intel Pentium 4 processor while running multithreaded Java applications.

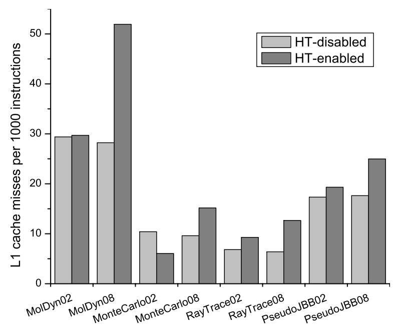

Figure 4. L1 data cache misses per 1,000 instructions

Figure 3 and Figure 4 examine the miss rates of trace cache and L1 data cache, respectively. With HT-disabled, the L1 data cache exhibits miss rates of 7–29 misses per 1,000 instructions. L1 instruction misses are even smaller, falling well below 2 trace cache misses per 1,000 instructions. The data also reveals that both trace cache and L1 data cache perform consistently worse while processor is running under SMT mode. For some cases, such as trace cache misses for RayTracer02, the number of misses is even doubled. Since the application code is identical, the causes of this problem are cache contention among threads and more OS activities.

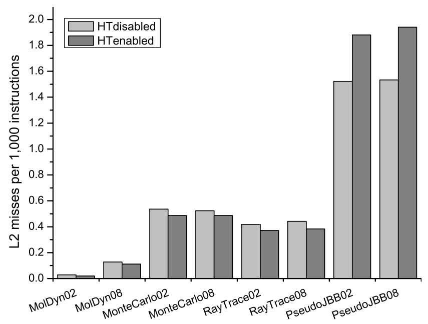

Figure 5. L2 cache misses per 1,000 instructions

The L2 cache miss rates are shown in Figure 5. We observe mixed behaviors for L2 cache while running multi-threaded Java applications. For three benchmarks (MolDyn, MonteCarlo and RayTracer), the L2 cache of the Pentium 4 performs better with Hyper-Threading turned on. This observation of is opposite to what we have seen in L1 caches. Combining the data of L1 and L2 caches, we believe that cache behaviors of Pentium 4 are related to Java memory footprint and cache sizes. Since the Pentium 4 has a very small L1 caches (12K *μop*s trace cache and 8KB data cache), the resource sharing will stress the L1 caches and cause severe conflicts. On the other hand, the size of L2 cache (1MB) is large enough to hold all data for most benchmarks even with two threads running simultaneously. Under this circumstance, the constructive cache interference will become dominant and thus improve L2 cache performance. The only exception, PseudoJBB, is caused by the extremely large memory footprint, which does not fit in the L2 cache. Its L2 misses increase with Hyper-Threading enabled because of contention.

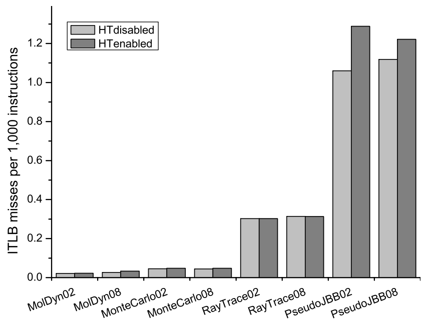

Figure 6. Instruction TLB (ITLB) misses per 1,000 instructions

Figure 6 presents the *Instruction TLB* (ITLB) misses we collected for multithreaded Java programs.

ITLB is responsible for translating instruction addresses into physical addresses to access the L2 cache when the machine misses the trace cache. The data of Figure 6 shows that ITLB performs slightly worse with Hyper-Thread turned on. For PseudoJBB, the miss rates increases significantly. Such degradation is caused by the ITLB design adopted by Intel. In the Pentium 4, the ITLB is partitioned among hardware contexts to support Hyper-Threading. Each logical processor has its own ITLB and its own set of instruction pointers to track the progress of instruction fetch. Therefore, the ITLB will perform worse than with HT-enabled because of partitioning.

Figure 7 shows the ratio of branches that miss in the *Branch Target Buffer* (BTB). The data clearly indicates that branch prediction of the Pentium 4 performs worse while running under Hyper-Threading mode. This is due to the fact that two logical processors share the same BTB. In SMT processors, conventional branch target prediction still drives instruction fetch, except being augmented by extra thread ID bits in the BTB. The Pentium 4 adopts this design and treats the BTB as a shared structure with entries that are tagged with a logical processor ID. This sharing will cause destructive interferences and thus increase BTB miss ratios.

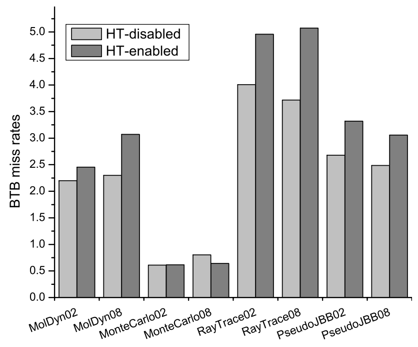

Figure 7. BTB miss ratios

4.2. Speedups of multiprogrammed Java applications

This section examines the performance of multiprogrammed Java benchmarks on the Pentium 4 processor. Multiprogrammed means running two independent applications simultaneously. Like [27], we use a variable, called *combined throughput*, to measure the performance of multiprogrammed Java applications. Combined speedup is defined as $C_{AB} = A^S / A^H + B^S / B^H$, where A^S and B^S are the execution times of program *A* and *B* running on a Hyper-Threading disabled processor, respectively. Likewise, A^H and B^H are the execution times of *A* and *B* running simultaneously on a Hyper-Threading processor. Measuring A^H and B^H is tricky due to staggered execution times. For instance, suppose the execution times of *A* and *B* are 3 seconds and 8 seconds, respectively. While running together, *A* probably finishes before *B*. If no precaution is taken during measurement, the execution time of *B* we measure will include the portion when *B* is running alone. For this purpose, we create a utility program to make *A* and *B* execute repeatedly [27]. The main process of utility program will re-launch the benchmark whenever either of them finishes. Each benchmark is repeated at least 12 times[1], and we drop the first run to exclude cold-start phase. The last run is also dropped to avoid uncompleted executions. The completion times of the remaining runs are then averaged as the execution time of each benchmark.

On a perfect time sharing machine without Hyper-Threading, the execution time of each benchmark will become half of the original value (i.e. $A^H = 0.5{*}A^S$ and $B^H = 0.5{*}B^S$). Therefore, C^{AB} is 1. On the other hand, a perfect SMP machine with two processors will allow program *A* and *B* to finish with the original execution time, which means $C^{AB} = 2$. Since SMT is a hybrid design, an SMT machine should provide a combined speedup between 1 and 2. Whenever C^{AB} is below 1, it implies a performance degradation for SMT machines.

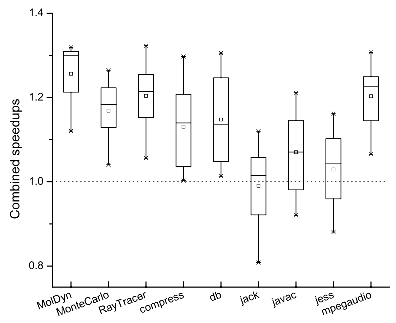

Figure 8. Distribution of combined speedup for multiprogrammed Java benchmarks

Figure 8 shows the combined speedups of cross-product of nine single threaded Java applications on Pentium 4 processor. In other words, each application is run with every other application and the speedup is measured. Due to the large volume of data, we use box chart to plot our data. The chart displays a summary of data distribution in quarterly percentiles. The middle line and the square in the box represent median and average speedups while a benchmark is running with other Java programs. The other two lines (edges of the

[1] The benchmark with smaller execution time probably has more than 12 runs to match up with the other benchmark.

box) in each direction show the 25th and 75th percentile, respectively. Two whiskers, which are lines that extend from the edges of the box to the observed minimum and maximum, are also shown in Figure 8.

The box chart clearly presents the properties of each benchmark while running with other programs. For instance, Figure 8 shows that MolDyn is one of the benchmarks that generally does not detriment the performance of other benchmarks. On average, it achieves a speedup of 1.26 while running with other Java programs. The best speedup, 1.32, is achieved while running RayTracer with MolDyn. On the other hand, many programs suffer greatly while running with jack. The average combined speedup of jack is below 1, which is an indication of performance slowdown on SMT processors. Compared with Figure 1, we see better improvement while running multiprogrammed applications. Such improvement is attributed to the behaviors of multiprogrammed benchmarks, whose performance is unaffected by many issues such as thread synchronization.

Figure 9 shows the individual performance of each benchmark in a multiprogrammed pair. The figure is organized using a color map with a black-white scale. Each square describes the combined speedup of the row benchmark when sharing the processor with the column benchmark [3]. The first impression of this figure is the symmetry. We observe a pretty good reflective symmetry in this figure, which means combining two programs *A* and *B* will have a speedup close to combing *B* with *A*. While intuitive, [3] reported that reflective symmetry is not very good while combining SPEC2000 programs on a Pentium 4 processor. Such a difference is caused by the scheduling policy of the OS which causes unfair sharing of execution time while running SPEC2000 programs.

Although most speedups are greater than 1, we do see some occurrences of slowdown in Figure 9. Out of 81 combinations, nine combinations are observed to have performance slowdowns ($C_{AB}<1$). As indicated in the figure (dash rectangle), these slowdowns are the combinations of three SPECjvm98 benchmarks (jack, javac, and jess). Likewise, other programs tend to perform worse while they pair with these three benchmarks. What really makes these bad partners for other programs? Analysis using performance counters reveals that the programs are experiencing greater cache misses (in trace cache and/or L1 data cache) while running with these three programs. The detailed statistical analysis revealed that trace cache is the major factor determining the pairing performance of Java applications. The complete results can be found in [11].

Due to the large number of combinations, we do not show the detailed micro-architecture behaviors of multiprogrammed Java benchmarks in this section. Our off-line data shows that the behaviors are similar to what we observed for multithreaded programs in section 4.1. Namely, SMT improves the performance of L2 cache and ITLB, while trace cache, L1 data cache and BTB perform worse.

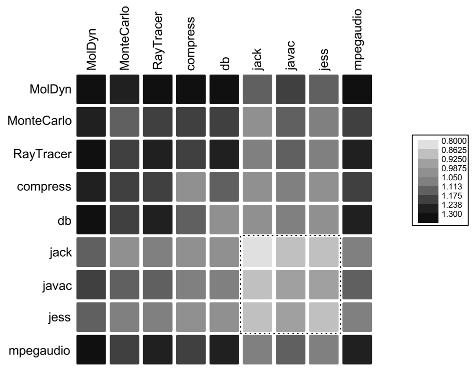

Figure 9. Combined speedup color map

4.3. Impacts of static resource partition on single-threaded application

Intel Hyper-Threading is an atypical SMT [28] design in that it statically partitions some hardware components, such as issue queue, ROB entries and load/store buffers, into two parts. Each part is assigned to one thread and can not be shared by the other thread. Under this design single-threaded programs might experience performance drops. To demonstrate this problem, we compare the performance of single-threaded Java applications running on a Pentium 4 processor with Hyper-Threading enabled and disabled (Figure 10).

Figure 10 shows that 7 out of 9 benchmarks are observed to have increased execution times, ranging from 0.15% to 62%, when Hyper-Threading is turned on. The results indicate that non-multithreaded Java applications do not benefit from the Hyper-Threading technology. In fact, they may incur a performance penalty because of static resource partitioning. For comparison, Figure 11 examines the speedups of Multiprogrammed benchmarks running on a Hyper-Threading processor. We run two identical copies of single threaded application simultaneously and measure the speedup. The results indicate that SMT can dramatically improve the performance of multiprogrammed benchmarks. The exception includes three benchmarks, whose behaviors have been discussed in Section 4.2.

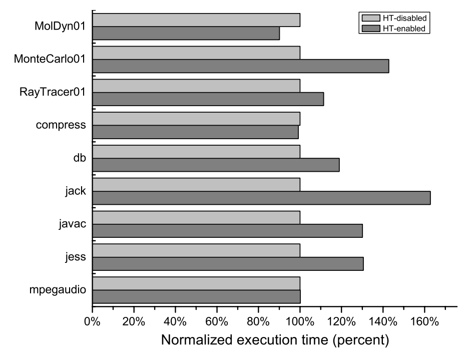

Figure 10. Impact of Hyper-Threading technology on single-threaded Java programs

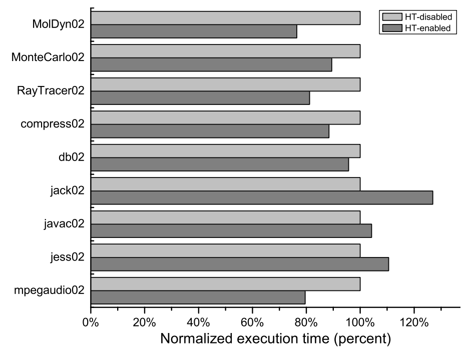

Figure 11. Impact of Hyper-Threading technology on multi-programmed programs

Since many Java programs are still single-threaded (i.e. one application thread), how to solve this performance slowdown is of great significance. Possible solutions exist on both hardware and software levels. The hardware solution is to allow the resources to be shared dynamically instead of partitioning them statically. When there is only one thread available for execution, this design will dedicate all hardware resources to this running thread and thus reach the optimal performance. Another solution, which does not require changing the architecture design, is to increase the thread-level parallelism (TLP) opportunity of single threaded Java programs on the software level. For example, the virtual machine might be able to use speculative pre-execution [1][5][4], a recently-proposed latency tolerance technology for the SMT architecture, to eliminate the performance degrading events of the main thread, such as cache miss and branch miss prediction. Because these events are the major performance bottlenecks of modern superscalar processors, this technique can improve the performance of Java programs.

4.4. Impacts of multithreading on Hyper-Threading processor

While previous research results indicate that eight hardware contexts are the optimal number for SMT processor design, the current Intel Pentium 4 only adopts 2 contexts. In Java multithreaded applications it is easy to have more than two threads running at the same time, especially for server applications. Therefore, understanding how the Java applications behave when the number of threads is greater than 2 is helpful for both software and hardware developers.

Using the same set of multithreaded benchmarks, we vary the thread number of each benchmark from 1 to 16 to observe the throughput changes (Figure 12). As might be expected, all benchmarks have dramatically increased throughputs when the number of threads increases from 1 to 2. This is attributed to better CPU resource utilization when there are two software threads running simultaneously. For most benchmarks, the IPC is not affected by the number of threads when there are more than 2. This suggests that two threads might be the optimal number for multithreaded Java programs running on current Hyper-Threading processors. The only exception is MolDyn, whose IPC drops significantly when the number of threads becomes 4. As shown in Figure 4, this is caused by substantially increased L1 data cache misses.

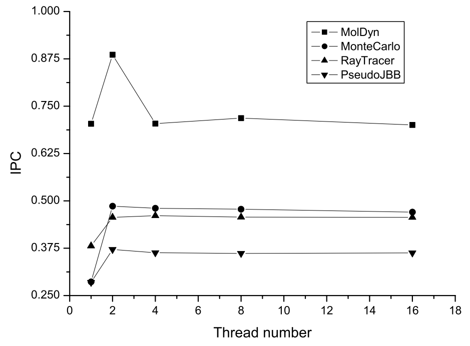

Figure 12. IPC vs. the number of threads

5. Conclusions

This study uses a mixture of multithreaded, multiprogrammed and single-threaded workloads to evaluate the performance of Java applications on Intel Hyper-Threading processors. A few key observations emerge. First, compared with other workloads, the performance improvement introduced by SMT for multithreaded Java programs is relatively small. The resource contention is the main reason for pipeline inefficiency. Secondly, due to the static partitioning design of Pentium 4 processors, most single threaded Java applications perform considerably worse when Hyper-Threading is turned on. These results indicate that programmers would better utilize the multithreading capability provided by Java programming language whenever possible. The JVM can also be improved to handle this problem by controlling Hyper-Threading execution more actively. Thirdly, multithreaded processors can effectively boost the performance of Java applications running together simultaneously. Most combinations we examined have good combined speedups. However, some Java programs show influence on the applications they paired with and slow down the overall execution time. Our offline analysis reveals that trace cache miss rate can be used to effectively predict the potential pairing performance.

Acknowledgements

Partial support for this work was provided by the National Science Foundation through grants CCR-0098235, CCR-0296131 and CCR-0219870 (to J. M. C.).

References

[1] Murali M. Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precoumputation. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, Göteborg, Sweden, June 2001.

[2] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 342–352, Tampa Bay, FL, October 2001.

[3] James R. Bulpin and Ian A. Pratt. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In *Proceedings of the 3rd Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, Munich, Germany, June 2004.

[4] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture (Micro)*, pages 306–317, Istanbul, Turkey, November 2001.

[5] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Hughes J. Christopher, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent prefetching by dependence graph precomputation. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, Göteborg, Sweden, June 2001.

[6] Manu Gulati and Nader Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA)*, San Jose, CA, February 1996.

[7] Samuel Z. Guyer and Kathryn S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*,Vancouver, Canada, October 2004.

[8] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal, and Wen-Mei W. Hwu. A study of the cache and branch performance issues with running Java on current hardware platforms. In *Proceedings of the 42nd IEEE International Computer Conference (CompCon)*, San Jose, CA, February 1997.

[9] Intel Corp. *VTune performance analyzer*. Available at `http://www.intel.com/software/products/vtune/`.

[10] Java Grande Forum. *The Java Grande Forum Multithreaded Benchmarks*. Available at `http://www.epcc.ed.ac.uk/javagrande/threads.html`.

[11] Wei Huang, Jiang Lin, Zhao Zhang, and J. Morris Chang. Towards Pairing Java Applications on Multithreaded Processors. *Department of Electrical and Computer Engineering Technical Report*, Iowa State University, 2005.

[12] Kaffe.org. *Kaffe virtual machine*. Available at `http://www.kaffe.org`.

[13] Martin Karlsson, Kevin E.Moore, Erik Hagersten, and David A. Wood. Memory system behavior of Java-based middleware. In *Proceedings of the 9th Annual International Symposium on High-Performance Computer Architecture (HPCA)*, Anaheim, CA, February 2003.

[14] Tao Li, Lizy K. John, Vijaykrishnan Narayanan, Anand Sivasubramaniam, Jyotsna Sabarinathan, and Anupama Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of the International Conference on Supercomputing (ICS)*, Santa Fe, NM, May 2000.

[15] Jack Lo, Luiz Barroso, Susan Eggers, Kourosh Gharachorloo, Henry Levy, and Sujay Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, Barcelona, Spain, June 1998.

[16] Yue Luo and Lizy K. John. Workload characterization of multithreaded Java servers. In *Proceedings of 2001 IEEE*

International Symposium on Performance Analysis of Systems and Software (ISPASS), Tucson, Arizona, November 2001.

[17] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, John Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technical Journal*, pages 4–15, February 2002.

[18] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Winter Technical Conference*, pages 29–41, San Diego, CA, January 1993.

[19] Ramesh Radhakrishnan, Vijaykrishnan Narayanan, Lizy K. John, and Anand Sivasubramaniam. Architectural issues in Java runtime systems. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, Toulouse, France, January 2000.

[20] Joshua Redstone, Hank Levy, and Susan Eggers. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, May 2000.

[21] Bohuslav Rychlik and John P. Shen. Characterization of value locality in Java programs. In *Proceedings of the 3rd Workshop on Workload Characterization (in association with ICCD)*, Austin, TX, September 2000.

[22] Brinkley Sprunt. Pentium 4 performance monitoring features. *IEEE Micro*, pages 72–82, July–August 2002.

[23] Brinkley Sprunt. *Brink and Abyss: Pentium 4 performance counter tools for Linux*. Available at `http://www.eg.bucknell.edu/bsprunt/emon/brink_abyss/brink abyss.shtm`.

[24] Standard Performance Evaluation Corporation (SPEC). *SPECjbb2000 Benchmark*. Available at `http://www.spec.org/osg/jbb2000/`.

[25] Standard Performance Evaluation Corporation (SPEC). *SPECjvm98 Benchmark*. `http://www.spec.org/osg/jvm2000/`.

[26] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (USENIX VM)*, San Jose, CA, May 2004.

[27] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New Orleans, Louisiana, September 2003.

[28] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, pages 191–202, Philadelphia, PA, May 1996.