Exploring Processor Design Options for Java-Based Middleware

Martin Karlsson and Erik Hagersten
Uppsala University
Department of Information Technology
P.O. Box 325, SE-751 05
Uppsala, Sweden
{martink, eh}@it.uu.se

Kevin E. Moore and David A.Wood
University of Wisconsin
Department of Computer Sciences
210 W. Dayton St.
Madison, WI 53706
{kmoore, david}@cs.wisc.edu

Abstract

Java-based middleware is a rapidly growing workload for high-end server processors, particularly Chip Multiprocessors (CMP). To help architects design future microprocessors to run this important new workload, we provide a detailed characterization of two popular Java server benchmarks, ECperf and SPECjbb2000. We first estimate the amount of instruction-level parallelism in these workloads by simulating a very wide issue processor with perfect caches and perfect branch predictors. We then identify performance bottlenecks for these workloads on a more realistic processor by selectively idealizing individual processor structures. Finally, we combine our findings on available ILP in Java middleware with results from previous papers that characterize the availibility of TLP to investigate the optimal balance between ILP and TLP in CMPs.

We find that, like other commercial workloads, Java middleware has only a small amount of instruction-level parallelism, even when run on very aggressive processors. When run on processors resembling currently available processors, the performance of Java middleware is limited by frequent traps, address translation and stalls in the memory system. We find that SPECjbb2000 differs from ECperf in two meaningful ways: (1) the performance of ECperf is affected much more by cache and TLB misses during instruction fetch and (2) SPECjbb2000 has more memory-level parallelism.

Keywords: Java, Middleware, workloads, ILP, CMP, Characterization

1 Introduction

The rapid rise of electronic commerce and Internet-delivered computing services has triggered explosive growth in the use of middleware. Simultaneously, Java has gained popularity as the programming environment of choice for Internet server software. Java-based middleware hosts the Internet presence of some of the world's largest companies in industries from financial services to airlines, making Java-based middleware one of the primary workloads for multiprocessor servers today.

The continued decrease in transistor size and the increasing delay of wires relative to transistor switching speeds has led to the development of chip multi-processors (CMPs) [2][23][27]. The introduction of CMPs presents new challenges and trade-offs to computer architects. In particular, architects must now strike a balance between allocating more resources to each processor against the number of processors on a chip. The proper balance of resources devoted to parallelism at the instruction level and parallelism at the tread level depends on the application. Previous studies have explored the limits to thread-level parallelism (TLP) and characterized the multiprocessor memory system behavior of Java middleware [13][14]. Here, we investigate the limits to instruction-level parallelism (ILP) and combine our findings with insights to TLP from previous papers to explore the design space of CMPs.

In this paper, we present a detailed characterization of two popular Java server benchmarks, ECperf and SPECjbb2000 (JBB). Where ECperf, also has been repackaged with only minor changes and released as SPECjAppServer2001 and SPECjAppServer2002. ECperf closely resembles commercially deployed Internet applications—it is a distributed system with clients, middleware and database all running separately. JBB emulates a 3-tiered application in a single process by simulating clients, and a database in the same Java virtual machine (JVM) as the middleware.

We attempt to classify Java middleware as a workload by comparing the behavior of ECperf to that of two other important commercial workloads, on-line transaction processing (OLTP) and static web serving (APACHE), and to several SPEC benchmarks. We also compare and contrast the behaviors of ECperf and JBB to see how well the simple

JBB approximates the more complicated ECperf.

We find that the behavior of ECperf is similar to that of other commercial workloads. Like OLTP and APACHE, ECperf exhibits limited ILP, even on aggressive processors. On very aggressive processors, its performance is limited by address translation and frequent traps. We notice two substantial differences between ECperf and JBB. JBB, like the SPEC CPU2000 benchmarks, is not limited by misses in the instruction cache or ITLB, and JBB has more memory-level parallelism.

The rest of this paper is organized as follows: Section 2 describes our simulation methodology. Section 3 presents the level of instruction-level parallelism (ILP) present in the workloads and identifies several limiting factors. Section 4 analyzes the bottlenecks present in a realistic microprocessor. Section 5 discusses how our results from sections 3 and 4 might influence architects seeking to optimize per-chip performance of Java middleware on a CMP. Section 6 outlines some related work in workload characterization. We present our conclusions in Section 7.

2 Methodology

All of the results in this paper were generated with full-system execution-driven simulation. We used full-system simulation because the operating system makes up a substantial portion of the runtime for our commercial workloads. Execution-driven simulation allowed us to model hardware that would be impossible to build. In particular, we simulated several processor configurations with one feature "idealized," or made perfect.

ECperf ECperf is a middle-tier benchmark designed to test the performance and scalability of a real 3-tier system. ECperf models an on-line business using a "Just-In-Time" manufacturing process (products are made only after orders are placed and supplies are ordered only when needed). It incorporates e-commerce, business-to-business, and supply chain management transactions.
SPECjbb2000 SPECjbb2000 is a server-side Java benchmark that models a 3-tier system, focusing on the middle-ware server business logic and object manipulation. The benchmark includes driver threads to generate transaction as well as an object tree working as a back-end. Our experiment use 24 driver threads (1.5 per processor) and 24 warehouses (with a total data size of approximately 500MB).
APACHE Static Web Content Serving: Apache with SURGE. We use Apache 2.0.43 configured to use a hybrid multi-process multi-threaded server model with 64 POSIX threads per server process. Our experiments use a hierarchical directory structure of 80,000 files (with a total data size of approximately 1.8 GB) and a modified version of the Scalable URL Reference Generator (SURGE [1]) to simulate 6400 users (400 per processor) with an average think time of 12ms.
OLTP On-Line Transaction Proccessing: DB2 with a TPC-C like workload. The TPC-C benchmark models the database activity of a wholesale supplier. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. Our experiments simulate 256 users (16 per processor)without think time. The simulated users query a 5GB database with 25,000 warehouses stored on eight raw disks and a dedicated database log disk.
SPEC CPU2000 We have chosen four benchmarks, GCC, BZIP, PERLBMK and MCF, from the SPEC CPU 2000 Integer suite [25]. GCC is an compilation with an aggressively optimized C compiler. BZIP is a compression and decompression of a TIFF image, a program binary, and a source tar file. PERLBMK is an PERL interpretor running a scripts for mail generation. MCF is a combinatorial optimization of a vehicle depot scheduling problem.

Table 1. Benchmark descriptions.

2.1 Workloads

To better understand the characteristics of our two Java server workloads, we compared them to two other commercial workloads, APACHE and OLTP, and four benchmarks from the SPEC2000 suite [25]. Table 1 presents a description of each application and the input parameters we used to run them. We ran both of our Java workloads on Sun's Hotspot 1.4.1 Server JVM and all of our workloads on Solaris 8.

2.2 Simulation Environment

For our simulations, we used the Simics full-system simulator [17]. Simics is an execution-driven functional simulator that models a SPARC V9 system accurately enough to run Solaris 8 unmodified. To model the timing of complex out-of-order processors we extend Simics with the Timing-first simulator, TFsim [20], and a detailed memory hierarchy simulator [19]. TFsim is a detailed timing simulator that models an out-of-order processor loosely based on the MIPS R10000 [28].

2.3 Measuring Performance Cost in Out-Of-Order Processors

In order to isolate the ILP limiting effects of various processor and memory system structures, we selectively idealize processor structures to approximate the cost of various aspects of instruction processing. The cost of a structure is defined as the difference between the performance with that structure idealized, or made perfect, and the performance of the base machine. Instead of measuring the costs of each structure individually, for simplicity, we measure the sum of the costs of various structures by idealizing structures such as caches and branch predictors in succession.

Our methodology allows us to model both perfect caches and perfect branch prediction on otherwise realistic processors. We model perfect branch prediction by recording the direction or target of all branches in a branch trace. We use separate traces for perfect branch direction prediction and perfect branch target prediction so that our timing simulator can execute arbitrarily far down a mispredicted branch capturing wrong-path effects. This separation enables us to distinguish the effect of idealizing the branch target buffer (BTB), return address stack (RAS) and the directional branch predictor. When simulating a perfect cache, which always hits, we still model each access throughout the memory hierarchy, which allows us to maintain approximately the same cache behavior in the unified lower levels of the memory hierarchy.

3 Limits on Instruction-Level Parallelism

Many modern microprocessors use out-of-order processing, branch prediction, non-blocking caches and other techniques to execute independent instructions in parallel. The number of instructions that can be executed in parallel is limited both by the characteristics of the workload and the resources available in the processor. In this section, we first estimate the amount of ILP available in each workload by eliminating as many constraints as possible in the hardware. This provides a generous estimate—albeit not a hard limit—of the speedup that can be achieved by increasing the size and complexity of each processor core. Next, we identify specific limitations on ILP in Java middleware.

3.1 Estimating the Level of ILP

To estimate the practical limit of ILP in these workloads, we simulate a 64-issue uniprocessor with both perfect caches and perfect branch prediction (i.e., the branch prediction is always correct and all memory accesses hit in the L1 cache). Our simulated processor has a five-stage pipeline, a 1024-entry reorder Buffer and infinite load/store queues. We measure the number of instructions executed per cycle (IPC) while increasing the size of the issue window.

Even this idealized processor is unable to exploit more than a small amount of ILP in any of our commercial workloads. As shown in Figure 1 (a), issue windows larger than 64 entries benefit only one of our commercial workloads, OLTP, on which performance peaks at a window size of 256 entries. Furthermore, much of the ILP exploited by our idealized processor is due to its unattainable perfect memory hierarchy. To isolate the effect of memory accesses on ILP, we simulate 2-cycle 4-way 32 KB L1 data cache, an 18-cycle 8-way 1 MB L2 cache and a 250-cycle memory latency.[1] Note that the instruction cache and branch predictor are still perfect. Comparing Figure 1 (a) and Figure 1 (b) highlights the effect of replacing the perfect memory sytem with a more realistic memory hierarchy. Performance improvement from increasing the window size is almost completely eliminated for all of the commercial workloads. Only PERL and BZIP continue to benefit from large instruction windows when memory accesses are modeled.

[1]This is the same memory system configuration as for the medium core shown in Table 6.

3.2 Traps: An ILP Limitation

Frequent traps, limit the benefits of large windows in out-of-order processors running Java middleware. On ECperf, we measured an average of 4.8 traps per 1000 instructions, which means that for a machine with a 1024-entry issue window there are, on average, more than 4 instructions that will cause a trap or exception in the window. Since exceptions are commonly detected at commit stage and usually require a pipeline flush, the cost of exceptions can be quite significant for processors with large issue windows. The effect of traps on ILP is not limited to Java workloads. We also find that the two applications whose performance scales the least with the issue window scaling, Apache and MCF, show some of the highest trap frequencies.

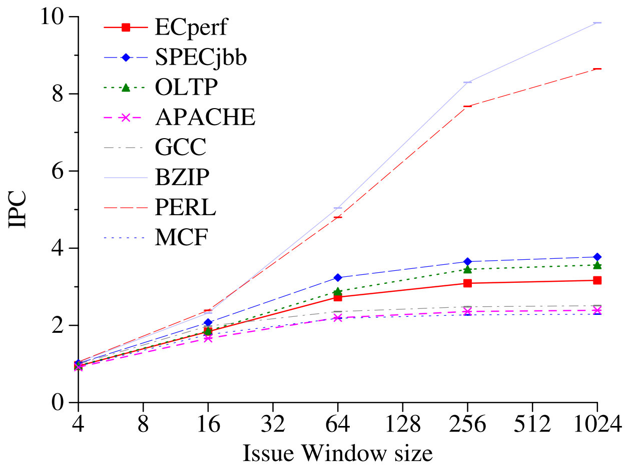
As shown in Table 2, the most frequent types of traps in Java workloads are TLB misses and register window spill/fill exceptions. Although the most common traps in our experiments are SPARC-specific—software-handled TLB misses and register window spill and fill traps—we believe that large instruction footprints and deep call stacks are inherent to Java middleware and will be an ILP limitation on other architectures as well.

Throughout this paper we have simulated a software-managed 128-entry four-way set-associative instruction and data TLB's and eight register windows. Unfortunately, since the TLB's and the register windows are part of the architected state in the SPARC architecture, our current simulator infrastructure does not allow us to idealize them.

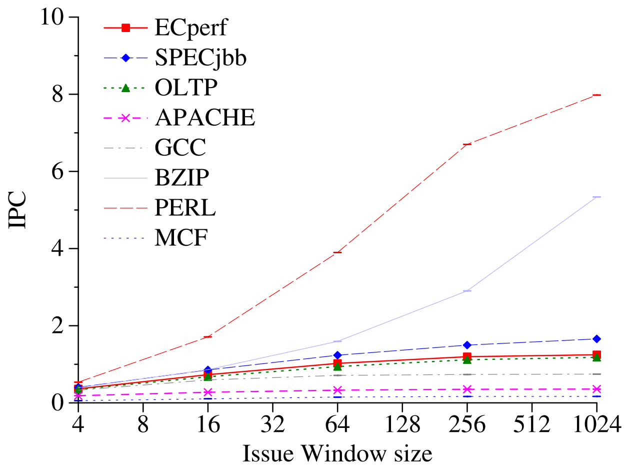
	DTLB	ITLB	Spill/Fill	Trap	Total
ECperf	0.99	2.08	1.73	0.04	4.84
JBB	0.22	0.07	0.13	0.05	0.47
OLTP	0.92	1.00	1.47	0.07	3.47
APACHE	1.99	0.43	2.38	0.13	4.94
GCC	0.01	0.00	0.06	0.00	0.07
BZIP	0.03	0.00	0.00	0.00	0.03
PERL	0.06	0.00	0.30	0.00	0.37
MCF	8.50	0.00	0.00	0.00	8.50

Table 2. The number of traps/exceptions taken per 1000 retired instructions

IEEE COMPUTER SOCIETY



(a) 64-issue machine with perfect caches



(b) 64-issue machine with 32KB L1D and 1MB L2

Figure 1. Estimating the available amounts of instruction level parallelism with and without a perfect data cache.

3.2.1 TLB Misses

TLB misses, especially ITLB misses, place a significant limitation on the ILP in ECperf. We observe an ITLB miss rate on ECperf (2.08 misses per 1000 instructions) that is twice as high as the miss rate for OLTP and roughly five times as that for APACHE. Since ITLB misses are extremely hard to overlap both with software or hardware managed TLBs, we identify the ITLB performance as a significant bottleneck to single-thread performance. We find that the DTLB miss rate for ECperf is on par with OLTP, which is known to have a significant execution stall time due to DTLB misses.

One optimization that improves the performance of TLBs for workloads with large instruction and data sets is the use of large pages. Our JVM, HotSpot 1.4.1, includes support for Intimate Shared Memory (ISM), a Solaris API that supports the use of 4 MB pages (normal pages in Solaris are 8 KB). In our configurations, however, HotSpot uses these pages for the heap, but not for code compiled with its just-in-time compiler. Therefore, when running ECperf, the DTLB has a much greater reach than the ITLB. Solaris 9 contains a new API, MPSS, which replaces ISM [21]. When run on Solaris 9, HotSpot uses MPSS to create large pages for both code and heap. We anticipate that we will see fewer ITLB misses in ECperf when we update our configuration to Solaris 9.

The negative pipeline effects from software handling of DTLB misses can be reduced by hardware managed TLB's or by inlining exception handlers, which avoids pipeline flushes and allows independent instructions to complete out of order with the faulting instruction as proposed by Jaleel and Jacob [11]. Alternatively, Zilles et al. [29] propose using idle threads in an multithreaded processor to handle exceptions thereby avoiding squash and re-fetch of the instructions following the faulting instruction. Note, however, that neither of these approaches can reduce the penalty of an ITLB miss.

3.2.2 Register Window Manipulation

We observe a substantial amount of spill/fill traps in ECperf, APACHE and OLTP. The register windows in the Sparc architecture cause spill/fill traps when the call stack depth exceeds the number of available register windows—the processors we simulate have 8 register windows. Although register windows are specific to SPARC, the call stack behaviour that generates spill/fill traps could also limit performance in a flat register architecture, where register content is manually saved on the stack between procedure calls. Given the Call/Return frequencies observed in our Java workloads, it is likely that several procedure calls would be in-flight at the same time, leading to artificial dependencies on the stack pointer, which could severly reduce the benefits of large out-of-order windows.

Moreover, if multiple procedure calls at the same call depth are in flight simultaneously, they will reuse the same stack space. Hence loads and stores associated with the first call will access the same stack addresses as loads and stores of the second call. If these accesses are executed out of order they may violate memory ordering rules leading to costly replay traps. Such traps, however, could potentially be avoided with memory dependence prediction [22][6].

In Java, performance inhibitors such as frequent traps, can be addressed in the virtual machine software as well as in the hardware. We propose one such optimization based on the following observation. We find that in ECperf the JVM only uses spill_1 traps, which spills a single register window. We also note that for 50% of the spill traps, three or more spill_1 traps occur in a row before a fill trap happens. The SPARC architecture provides different spill/fill traps for spilling/filling multiple register windows at a time. For example, the Ultrasparc III can spill or fill one, three or six register windows at a time. Our results indicate that 99 % of the spill traps in ECperf occur from only 15 different user-level instructions. If these trap locations could be indentified and changed statically or dynamically to spill_3 traps, the number of pipeline-disrupting spill traps could be decreased by a third. We observe the same behavior for fill traps as well.

3.3 MLP and Memory System Impact

Memory-level parallelism (MLP) is an important workload characteristic because it measures the ability of the processor to overlap the latency of multiple cache misses. As latencies continue to increase, the importance of exploiting MLP becomes more and more important.

We estimate the amount MLP in our workloads by counting the number of outstanding cache misses each time a cache miss occurs. We perform these measurements using our most aggressive processor (1024-entry issue window) from Section 3.1 and Figure 1 (b) in order to maximize the MLP.

Since the number of outstanding misses may vary significantly during execution, we present the maximum number of outstanding cache misses recorded during various fractions of the total misses in addition to the average number of outstanding misses. Table 3 shows the highest number of outstanding misses during the 95% and 99% of misses with the least number of already outstanding misses—e.g. for ECperf the 99% column indicates that only 1% of the misses occurred when more than 5 L1 data cache misses were already outstanding.

All of the commercial benchmarks except OLTP exhibit a similar level of MLP. On average, OLTP has 8% more outstanding L2 misses than JBB and 20% more than ECperf. For the commercial workloads six MSHR's for the L1 data cache would be sufficient during 99% of the execution of all the commercial workloads.

	L1 Data Cache			L2 Unified Cache		
	95%	**99%**	**Avg.**	**95%**	**99%**	**Avg.**
ECperf	2	5	1.73	2	5	1.71
JBB	3	5	1.99	3	5	1.91
OLTP	3	6	1.69	4	11	2.06
APACHE	3	5	1.53	3	5	1.52
GCC	1	1	1.57	0	1	1.44
BZIP	15	38	5.30	9	31	4.39
PERL	0	1	1.43	0	1	1.43
MCF	3	16	4.82	3	16	4.69

Table 3. Estimating the degree of memory level parallelism. Maximum number of outstanding misses recorded for 95 and 99% of the miss events with a 1024 entry issue window.

4 Bottleneck Analysis

In order to find, quantify and isolate the limitations to ILP for a more realistic machine, we measured the IPC while idealizing structures like caches and branch predictors. In this section, we model a more conservative 3-issue processor, with a single load/store unit. We select this conservative design because our ILP limitation study in Section 3 reveals that even a processor with a very high issue width and large instruction window is unable to exploit any significant degree of ILP for our Java workloads. We also find that even more modest increases in issue width produce only small performance improvements over our 3-issue processor—e.g. an 8-issue processor with 2 load/store units only gave a 21% speedup for ECperf.

The simulated machine has the baseline memory system (2 cycle 4-way 32 KB L1 caches and an 18-cycle 8-way 1 MB L2 cache). Our baseline branch predictor are comprised of a 2.5 KB YAGS directional predictor [8], a 256-entry cascaded indirect predictor [7] and a 16-entry return-address stack [12].

For the experiments in this section, we also extend the pipeline depth to 16 stages to more accurately model pipeline effects. Our choice of 16 stages is a result of a simplistic optimal pipeline depth analysis similar to the one described by Hartstein et al. [10]. We determine the optimal pipeline depth by measuring the time per instruction while we increase the pipeline depth by scaling the decode, schedule and execute stages linearly—i.e., we add one additional stage at a time to each phase. In our model, a 22-stage pipeline minimizes the time per instruction. However, we conservatively choose to model a 16-stage pipeline because the improvement from 16 to 22 was negligible.

We measured the cost of branch mispredictions and

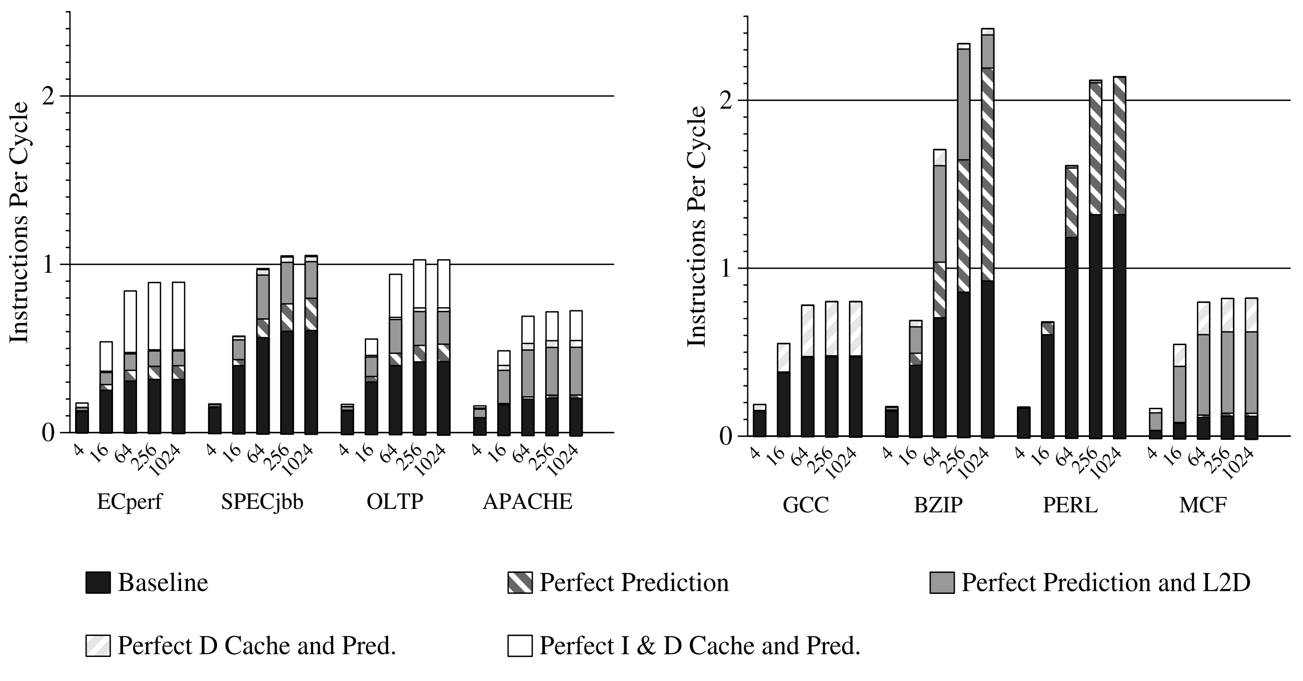


Figure 2. Performance improvements when idealizing certain structures compared to a 3-issue baseline configuration (Medium core), while scaling the issue window.

misses to the instruction and data caches by comparing the performance of our baseline machine to a similar machine with that one particular feature (a branch predictor or cache) made perfect. For example, we measure the cost of data misses to the unified L2 cache by simulating a perfect second-level cache[2] for all data accesses.

Figure 2 displays the performance improvement derived from idealizing more and more structures. Starting with a baseline design and then first idealize the branch predictors and then idealizing more and more of the memory hierarchy. The height of the bar represents the performance obtained when all structures are perfect.

4.1 Performance Effect of Cache Misses

We find that the performance of ECperf is more dependent on the instruction cache than any of our other benchmarks, which matches the high instruction cache miss rate reported by Karlsson et al. [13]. For ECperf, idealizing the L2 data accesses produces a considerable performance improvement. Idealizing the L1 data cache in addition, however, does not improve performance any further. We also observe that increasing the size of the instruction window beyond 64 entries does not significantly improve performance on either our baseline or idealized processors for any of our commercial applications. However, the SPEC applications BZIP and PERLBMK continue to benefit from larger instruction windows.

[2]Note that a perfect cache is not the same as an infinite cache as a perfect cache hits even for cold miss.

Figure 2 illustrates the degree to which each structure limits ILP for a particular workload, however, it does not indicate how much those limitations can be reduced by improving the structure. In order to test the feasibility of improving the caches, we measure the relative performance improvement of achievable optimizations such as doubling the cache size or associativity.

	L1 Data Cache		L1 Instr Cache	
Application	64KB	8-w	64KB	8-w
ECperf	0%	0%	0%	0%
JBB	0%	0%	0%	0%
OLTP	0%	0%	1%	0%
APACHE	2%	2%	3%	2%
GCC	0%	0%	0%	0%
BZIP	0%	0%	0%	0%
PERL	0%	0%	0%	0%
MCF	0%	0%	0%	0%

Table 4. Performance improvement over a 64-entry issue window baseline configuration when doubling cache size or associativity.

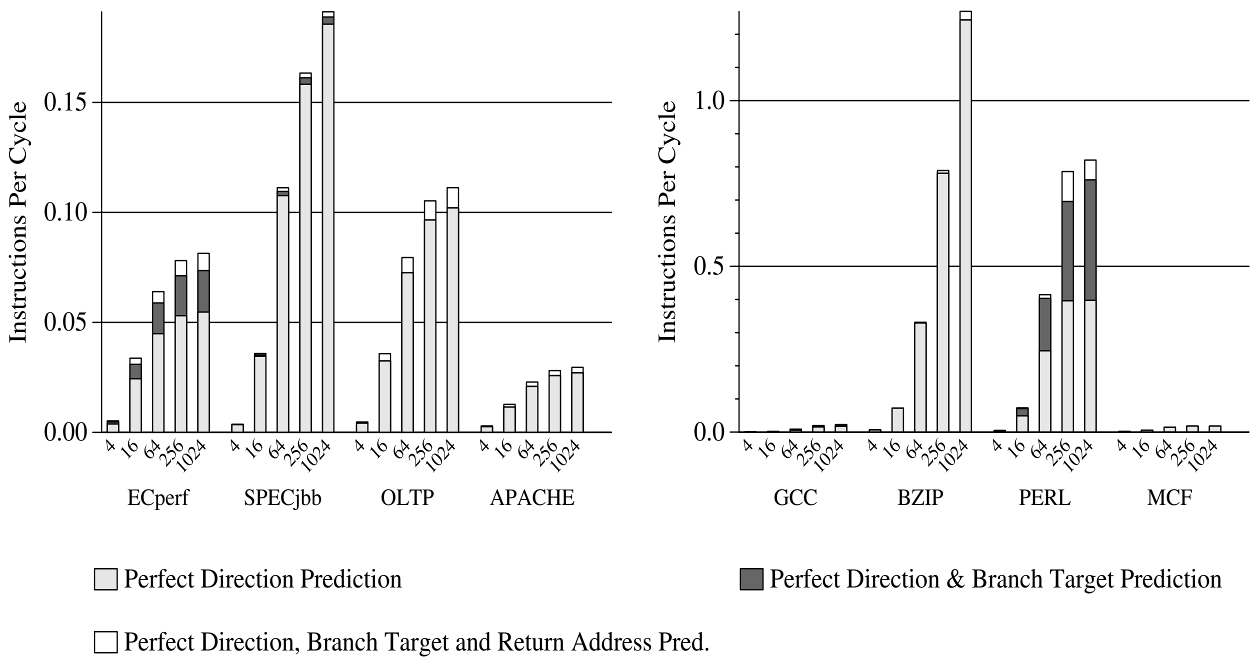


Figure 3. Performance improvement with perfect branch predictions structures compared to a 3-issue baseline configuration. Note this is a scaled up breakdown of the perfect prediction improvement stack from Figure 2.

Not surprisingly, doubling the size and associativity of the caches substantially reduces the cache miss rate. For example, when doubling the cache size, the number of data cache misses for ECperf and OLTP are reduced by 17% and 16% respectively. However, Table 4 shows that there is no corresponding improvement in performance for any application except APACHE. This lack of performance improvement indicates that L1 cache misses are not a primary bottleneck for these workloads running on our baseline processor.

4.2 Performance Effect of Branch Prediction

Modern processors use several different mechanisms to predict the instruction stream, including branch direction prediction, branch target prediction and return address prediction. We investigate breakdown the cost of several common types of mispredictions by measuring the effect of idealizing different types of predictors over our baseline branch predictor configuration.

As we can see in Figure 3, the bulk of the prediction cost in performance for all workloads is due to mispredictions of conditional branches. For ECperf, also perfecting the branch target buffer and return address stack yields a noticeable performance improvement. The same observation can be made for PERL. Perfect prediction improves the relative performance of ECperf by 21%, which is more than any of the other commercial benchmarks.

	Branch Predictors			
Application	Direction	BTB	RAS	All Perfect
ECperf	4%	0%	0%	21%
JBB	2%	0%	0%	19%
OLTP	6%	0%	0%	18%
APACHE	5%	2%	2%	11%
GCC	1%	0%	0%	5%
BZIP	1%	0%	0%	46%
PERL	3%	2%	0%	35%
MCF	0%	0%	0%	11%

Table 5. Performance improvement over a 64 entry issue window baseline configuration when doubling predictor sizes.

As above, we estimate how much of the possible performance improvement is attainable with realistic hardware by measuring the overall IPC improvement over the baseline when doubling the size of the different predictors. For all applications, especially the commercial ones, we observe significant gains from doubling the direction predictor. Doubling the BTB or the RAS only improves performance for APACHE and PERL.

Processor Core	LARGE	MEDIUM	SMALL
Fetch/Issue/Commit	8	3	1
Branch Pred.	10KB	2.5KB	0.6KB
BTB	1024	256	64
Return Address Stack	32	16	8
Mispred. Penalty	13	13	13
L1 I-Cache (4-way)	64KB	32KB	16KB
L1 D-Cache (4-way)	64KB	32KB	16KB
L1 Latency (I & D)	3 cycles	2 cycles	2 cycles
L2 Unified (8-way)		1MB	
L2 Latency		18 cycles	
Memory Latency		250 cycles	

Table 6. Processor Core configurations.

For ECperf, despite a 10% decrease in the BTB misprediction rate, we observe no noticeable improvement in performance. We hypothesize that this discrepancy is is due to the fact that programmers and compilers often put save and restore instructions in the delay slot of call and return instructions. When a spill or fill trap occurs on such an instruction, the pipeline is flushed. Therefore, any mispredictions immediately preceding the trap will have little or no effect on performance.[3]

5 Chip Multiprocessor Trade-offs

In order to make a back-of-the-envelope estimation of the CMP design that is best suited for Java-middleware workloads, we compare the performance of three processor configurations (small, medium and large) that represent different ILP vs. TLP design points. We measure the performance of each processor with three different L2 cache sizes. The relative performance improvement from each increase in processor size provides a rough estimate of the amount of additional chip area or power consumption that may be justified by the corresponding increase in throughput.

Our three processor configurations, large, medium and small, are described in Table 6. The medium core parameters were selected based on our earlier findings to represent a modestly aggressive processor in terms of single-thread performance, and relatively small in order to fit as many as possible on a die. It has the baseline memory system and branch predictor configurations. The small and large core designs was chosen as reference points in terms of single-thread performance and size.

The large and medium cores have issue windows with 128 and 32 entries, respectively, while the small core is single-issue and in-order. The issue width and the size of the L1 caches and predictors are increased by at least a factor 2 between each core. We also simulate each core with 1, 2 and 4 MB caches to observe each core's performance dependence on the L2 cache size. Previous studies on ECperf have shown both that ECperf scales linearly up to 8 processors and that a few MB of cache captures the entire working set [13][14] when shared by 8 threads. Based on these observations, we assume linear scaling on ECperf when comparing CMP designs.[4]

For ECperf, we notice a performance speedup of 2.5 times between the small and medium core, but only a 21% improvement from medium to large. Therefore, the large core can consume at most 21% more area than the medium core for it to be as efficient from a throughput-per-chip-area standpoint for this particular workload. For the medium core, we note a 14% improvement when increasing the L2 size from 1MB to 2MB, but only an additional 11% when increasing from 2MB to 4MB. Taking the area trade-off one step further, we compare the performance effect of adding cores instead of cache. For the medium core on ECperf, this implies that since we assume linear scaling when increasing the number of cores. Increasing the L2 cache size from 2MB to 4MB must increase the area by no more than 11% to be more beneficial than adding additional cores for ECperf.

When power is taken into consideration, cache would be favored instead of cores since caches consume significantly less power (both in terms of leakage and dynamic power) than processor cores. Adding cache can also reduce memory bandwidth consumption, which could become a major performance limitation in large CMPs as more cores on a die will lead to fewer available pins per core.

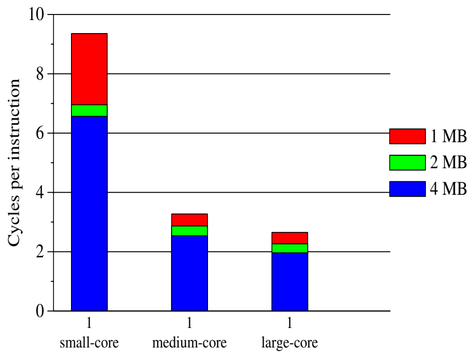
6 Related Work

Several previous papers have also analyzed Java workloads. Li and John identify the large number of branch sites in Java programs as the cause of poor branch prediction on these workloads [15]. Luo and John present an analysis of two Java server benchmarks (JBB and VolanoMark). They find that Java server workloads have poor instruction cache behavior and high ITLB miss rates [16]. Shuf et al. observe that prefetching is ineffective on Java workloads due to high TLB miss rates. Other studies characterize the memory system behavior of Java server workloads [4][5][13][18][26].

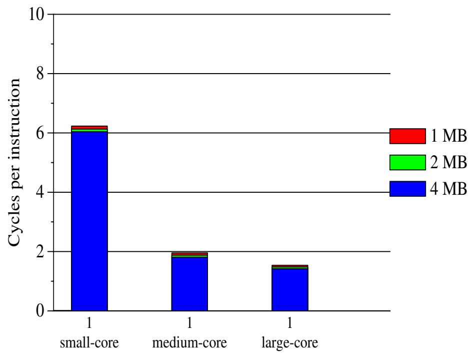
Many previous papers have studied the behavior of other commercial workloads. For example, Barroso et al. studied the memory system of an OLTP workload [3]. Redstone

[3]We assume an aggressive trap mechanism that does not squash the delay slot on a misprediction.

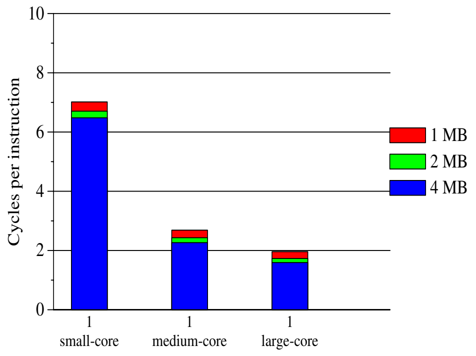
[4]Note that the linear scaling assumption does not hold for SPECjbb2000, APACHE and OLTP, since their dataset is larger than the simulated L2 cache sizes leading to potentially negative L2 sharing effects. We include them as uniprocessor performance reference points.



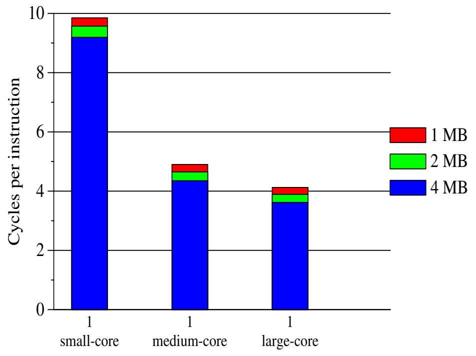
(a) ECperf



(b) JBB



(c) Apache



(d) OLTP

Figure 4. Chip trade-offs, Large cores vs. smaller cores and cores vs. cache

et. al. studied the operating system behaviour on an SMT architecture using APACHE [24].

Ekman and Stenström studied the performance and power impact of issue-width in CMP cores using the SPLASH-2 benchmark suite [9] and found that CMP designs with fewer wider-issue cores perform as well with a comparable power consumption as designs with larger numbers of smaller cores.

7 Conclusions

Java middleware is an important and growing workload for server processors. This paper is the first to characterize this emerging workload using the step-by idealization of processor structures. We use this technique to illustrate the behavior of ECperf and JBB. We offer greater insight into the behavior of our Java workloads by comparing them to other well-known commercial workload and to several of the widely studied SPEC benchmarks.

We find that, at the processor level, Java middleware behaves much like other well known commercial workloads—ILP is limited by memory system stalls, branch mispredictions and frequent traps. For ECperf, as for other commercial workloads, instruction fetch is a potential performance bottleneck. Overall performance is limited by instruction cache and TLB misses for ECperf, OLTP and APACHE. Improving ITLB performance, perhaps by using large pages for Just-In-Time compiled code, is essential to the effective utilization of aggressive processors on ECperf.

For ECperf, we find that a modestly aggressive medium-size processor core achieves very close to the performance obtained by a more aggressive core. Our most aggressive processor achieved only an 21% speedup over our medium-size core despite having twice the issue width, amount of L1 cache and branch predictor state. Extremely simple processors, on the other hand, sacrifice a significant amount of easily exploitable ILP. Our medium-size processor outperformed a simple in-order processor by a factor of 2.5. Therefore, we believe that from a performance-per-engineer-year, performance-per-mm2 and performance-per-watt point of view modestly aggressive processor cores may be the best CMP design choice for Java middleware.

8 Acknowledgments

We would like to thank Håkan Zeffer, Greg Wright, Carl Mauer, Alaa Alameldeen and Milo Martin for valuable input and feedback on this work. This work is supported in part by the PAMP research program, supported by the Swedish Foundation for Strategic Research and by the National Science Foundation (CDA-9623632, EIA-9971256, EIA-0205286, and CCR-0324878), a Wisconsin Romnes Fellowship (Wood), and donations from Intel Corp. and Sun Microsystems, Inc. Dr. Wood has a significant financial interest in Sun Microsystems, Inc.

References

[1] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[2] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[3] Luiz A. Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, 1998.

[4] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An Architectural Evaluation of Java TPC-W. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 229–240, 2001.

[5] Nirut Chalainanont, Eriko Nurvitadhi, Kingsum Chow, and Shih-Lien Lu. Characterization of L3 Cache Behavior of Java Application Server. In *Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-7)*, 2004.

[6] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*, pages 142–153, 1998.

[7] Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 167–178. IEEE Computer Society, 1998.

[8] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–77. IEEE Computer Society Press, 1998.

[9] M. Ekman and P. Stenström. Performance and Power Impact of Issue-width in Chip-Multiprocessor Cores. In *Proceedings of International Conference on Parallel Processing (ICPP)*, 2003.

[10] A. Hartstein and T. R. Puzak. The Optimum Pipeline Depth for a Microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'01)*, pages 7–13, 2002.

[11] A. Jaleel and B. Jacob. In-line interrupt handling for software-managed TLBs. In *Proceedings. 2001 International Conference on Computer Design*, pages 62–67, Sept 2001.

[12] Stephan Jourdan, Tse-Hao Hsing, Jared Stark, and Yale N. Patt. The Effects of Mispredicted-Path Execution on Branch Prediction Structures. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 58. IEEE Computer Society, 1996.

[13] M. Karlsson, K. Moore, E. Hagersten, and D. A. Wood. Memory System Behavior of Java-Based Middleware. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Anaheim, California, USA, February 2003.

[14] Martin Karlsson, Kevin Moore, Erik Hagersten, and David Wood. Memory Characterization of the ECperf Benchmark. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002), held in conjunction with the 29th International Symposium on Computer Architecture (ISCA29)*, Anchorage, Alaska, USA, May 2002.

[15] Tao Li, Lizy Kurian John, and Jr. Robert H. Bell. Modeling and Evaluation of Control Flow Prediction Schemes Using Complete System Simulation and Java Workloads. In *10th IEEE International Symposium on Modeling Analysis Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, October 2002.

[16] Yue Luo and Lizy Kurian John. Workload Characterization of Multithreaded Java Servers. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2001.

[17] P. S. Magnusson, M. Christensson, D. Forsgren J. Eskilson, G. Hllberg, J. Hgberg, A. Moestedt F. Larsson, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, February 2002.

[18] Morris Marden, Shih-Lien Lu, Konrad Lai, and Mikko Lipasti. Memory System Behavior in Java and Non-Java Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, 2002.

[19] Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Bandwidth Adaptive Snooping. In *HPCA*, pages 251–262, 2002.

[20] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116. ACM Press, 2002.

[21] Sun Microsystems. Performance documentation for the java hotspot vm. http://java.sun.com/docs/hotspot/ism.html.

[22] Andreas Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin-Madison, 1998.

[23] Kunle Olukotun, Basem A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[24] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. *SIGPLAN Not.*, 35(11):245–256, 2000.

[25] SPEC. SPEC cpu2000 Page. http://www.spec.org/osg/cpu2000/.

[26] Lawrence Spracklen, Yuan Chou, and Santosh G. Abraham. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[27] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliam, and S. S. Tse. The MAJC architecture: a synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[28] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[29] Craig B. Zilles, Joel S. Emer, and Gurindar S. Sohi. The Use of Multithreading for Exception Handling. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–229, November 1999.

IEEE COMPUTER SOCIETY