Memory Characterization of the ECperf Benchmark

Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A.Wood

Uppsala University
Information Technology
Department of Computer Systems
P.O. Box 325, SE-751 05 Uppsala, Sweden
Email: {martink, eh}@it.uu.se

University of Wisconsin
Department of Computer Sciences
1210 W. Dayton St.
Madison, WI 53706
Email: {kmoore, david}@cs.wisc.edu

Abstract

The quest for quantitative performance analysis in computer architecture research and design led to many research papers based on the commercially important DSS and OLTP database applications. Today's middleware, and in particular application servers, are rapidly gaining popularity as a new workload for commercial multiprocessor servers. Therefore, it is important to expand architecture studies into this area. SPEC has recognized the importance of application servers and Java-Based middleware with its recent adoption of SPECjAppServer2001, formerly ECperf, as one of its benchmarks. In this paper, we present a detailed characterization of the memory system behavior of ECperf running on both commercial server hardware and in a simulated environment. We find that the working sets of ECperf fit in a reasonably sized L2 cache, but that the miss rate is very sensitive to associativity. Furthermore, many of the L2 misses in ECperf are satisfied by data from the cache of a neighboring processor. Based on these findings, we conclude that ECperf is an excellent candidate for shared-cache memory systems, even when the total cache size is limited.

1 Introduction

The rapid growth of internet-delivered services has brought with it a new type of server workload, the application server. According to leading market research companies Gigaweb Corporation and International Data Corporation, the application server market grew 39% to 2.2 billion dollars in 2001 following several years of +100% growth. Although smaller than the market for databases (approximately 8.8 billion dollars in 2000), the large market for application servers is evidence that they are an important workload for today's servers. Several previous papers have characterized the performance of other commercial workloads, most notably database systems, running the TPC benchmarks [1][2]. Application servers, however, behave quite differently from database engines; they have much smaller data sets than databases and generate much less disk traffic. Also, many commercial application servers are implemented in Java. Like other Java workloads, they are influenced by the performance characteristics of the Java Virtual Machine (JVM) on which they run, including automatic garbage collection and Just-In-Time compilation. Unlike known Java benchmarks, application servers are optimized commercial applications.

We present a detailed characterization of the ECperf benchmark run on a leading commercial application server. ECperf is an appropriate benchmark for studying application server performance, because it models a 3-tiered system, and emphasizes the middle tier. Furthermore, in ECperf, each tier is run on a separate machine, making it easy to monitor the memory behavior of the application server separately. SPEC recently acknowledged ECperf by incorporating it into its suite of benchmarks as SPECjAppServer2001. We used a leading commercial application server to run ECperf on two realistic hardware configurations. We measured the effect of scaling ECperf on the memory systems of two multiprocessor servers. We compliment our scaling results with cache miss rates from a full system simulation of a shared-memory multiprocessor running ECperf. Finally, based on our findings, we discuss the architectural implications of this workload.

2 Background

At the center of modern web-services applications is the ability to connect web pages to databases. For data-

This work is supported in part by the National Science Foundation, with grants EIA-9971256, CDA-9623632, and CCR-0105721, the PAMP research program supported by the Swedish Foundation for Strategic Research, a Wisconsin Romnes Fellowships (Wood), and donations from Compaq Computer Corporation, Intel Corporation, IBM, and Sun Microsystems.

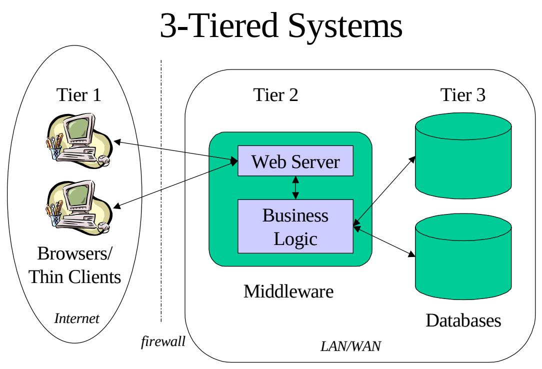

Figure 1 3-Tiered Systems

bases, connection to the web allows users to access data without installing a client program; for web pages, databases provide dynamic content and permanent storage. The software that implements the database-to-web connection is known as "middleware." Middleware can be stateless like a PERL script or Java Servlet, or a set of component classes embedded in an application server that stores the session information for its clients locally. Component-based middleware hosted by application servers reduce the burden on the database by not requiring additional database transactions to manage session information, and allow more complicated session data than is practical to pass with every web request. Application servers provide communication with both back-end databases and front-end web clients in addition to providing a framework to connect application specific "business rules," that govern the interaction between the two.

Today's web-services applications are deployed on heterogeneous systems arranged in distinct tiers, each of which has its own performance and state persistence requirements. Database systems have traditionally been built in a two-tiered, or Client/Server, architecture. That two-tiered approach is not well suited to the web because web browsers communicate in HTML and database servers communicate in SQL. Rather than add an HTML translator to an already complicated database server, systems designers have added an additional tier between the client and server (see Figure 1). This middle tier is responsible for implementing presentation logic and business rules, both of which are specific to the given application. Such systems are called "3-Tier." The first tier, or Tier 1, is the client, usually a web browser. The second tier or, Tier 2, is an application server that hosts application-specific middleware. The third tier, or Tier 3, is the data store, a highly reliable persistent storage mechanism. This tier is typically comprised of a single, or group of Database Management Systems (DBMS).

Recently, web services have been deployed in an "N-Tier" architecture in which the presentation logic is separated from the business rules. The presentation logic can be implemented by stateless servers and is sometimes considered to be a first-tier application. N-Tiered architectures allow the application server to focus entirely on the business logic.

2.1 ECperf Overview

ECperf is a middle-tier benchmark designed to test the performance and scalability of application servers and the computer systems that run them. It incorporates e-commerce, business-to-business, and supply chain management transactions. The presentation layer is implemented with Java Servlets and the business rules are built with Enterprise Java Beans.

In order to make the benchmark realistic, the authors of ECperf modeled 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). The application is divided into four domains, which manage separate data and employ different business rules [11].

Customer Domain

The Customer Domain models orders and customer interaction. Customers can create new orders, make changes to existing orders, and inquire about the completion status of their orders. The customer interactions are similar to On-Line Transaction Processing (OLTP) transactions.

Manufacturing Domain

The Manufacturing Domain implements the "Just-In-Time" manufacturing process. As orders are filled, the status of customer orders and the supply of each part used to fill the order are updated.

Supplier Domain

The Supplier Domain models interactions with external suppliers. A supplier is chosen for each purchase based on the number and type of the parts being ordered and the required delivery date. The parts inventory is updated as purchase orders are filled.

Corporate Domain

The Corporate Domain tracks customer, supplier and parts information. Most importantly, the Corporate Domain stores and updates credit information for each customer.

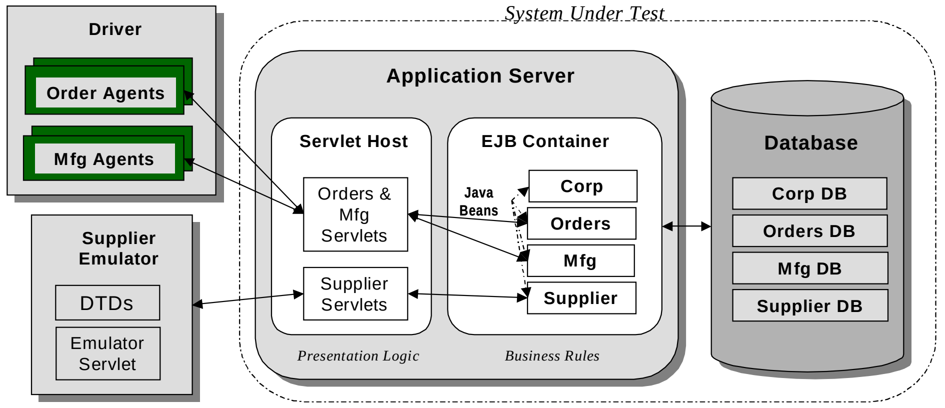

Figure 2 ECperf Setup

The ECperf specification supplies the Enterprise Java Beans that form the core of the application. These components implement the application logic that controls the interaction between orders, manufacturing and suppliers. In particular, that interaction includes submitting various queries and transactions to the database, and exchanging XML documents with the Supplier Emulator.

There are four parts of the ECperf benchmark, each of which is intended to run on a separate machine, or group of machines. Each of these parts is represented by a box in Figure 2 The first part, shown in the center of Figure 2, is comprised of the application server and ECperf Java Beans. Together, they form the middle tier of the system, which is the most important component to performance on ECperf. The next most important part of the system, in terms of performance, is the database. Though ECperf does not overly stress the database, it does require the database to keep up with the application server and to provide atomic transactions. The Supplier Emulator is implemented as a collection of Java Server Pages (JSP) hosted in a separate web container. Finally, the driver, which models customers and manufacturers, is a Java program that spawns several agents, each of which is a separate Java program. The driver coordinates these agents with Java Remote Method Invocation. Each high-level action in ECperf, such as a customer making a new order, or a manufacturer updating the status of an existing order is called a "Business-to-Business Operation," or "BBop." Performance on ECperf is measured in terms of BBops/minute. Official results include both the absolute throughput in BBops/min and the cost adjusted throughput in $/(BBops/min). The System Under Test (SUT) for the benchmark includes the application server (or servers), the database and the connection between them. The driver and Supplier Emulators, which model the behavior of customers and external suppliers respectively, are not part of the SUT and are therefore not included in the benchmark measurements or cost calculations. ECperf has recently been adopted by the Standard Performance Evaluation Corporation (SPEC) and will be released as SPECjAppServer2001 [12].

2.2 Enterprise Java Beans

ECperf is implemented using Enterprise Java Beans (EJB), a part of the Java 2 Enterprise Edition (J2EE) standard. Like ordinary Java Beans, EJB are reusable Java components. However, EJB are server-side objects intended to be the building blocks for web-service applications. EJB, are merely components, and do not make an application by themselves. These components are not useful until they are deployed on a J2EE compliant application server. Inside the server, an EJB "container" hosts the beans and provides important services. In particular, EJB rely on their containers to manage connections to the database, control access to system resources, and manage transactions between components. Often the container is also responsible for maintaining the persistent state of the beans it hosts. The application server controls the number of containers and coordinates the distribution of client requests to the various instances of each bean.

2.3 Java Application Servers

To host ECperf, we used a leading commercial Java-based Application Server. That server can function both as a framework for business rules (implemented in EJB) and host presentation logic, including Java Server Pages (JSP) and Java Servlets, as well as static web pages. As an EJB container, it provides required services such as database connections and persistence management. It is also designed to scale up to support large numbers of connections and must therefore also provide better performance and scalability than a naïve implementation of the J2EE standard, such as the Reference Implementation included in the Java 2 Enterprise Edition Development Kit. Two important performance features of our particular server are thread pooling and database connection pooling. The application server creates a fixed number of threads and database connections, which are maintained as long as the server is running. The application server allocates idle threads or connections out of these pools instead of creating new ones and destroying them when they are no longer needed. Database connections require a great deal of effort to establish and are a limited resource on many database systems. Connection pooling increases efficiency because many fewer connections are created and opened. In addition, connection pooling allows the application server to potentially handle more simultaneous client sessions than the maximum number of open connections allowed by the database at any time. Thread pooling accomplishes the same conservation of resources at the Operating System level that database connection pooling does at the database. Our experience tuning the application server showed that configurations with too many threads spend much more time in the kernel than those that are well tuned. Another performance feature of our application server is component caching. Bean instances are cached saving database queries and memory allocations.

3 Hardware Monitoring Experiments

We have used an Oracle Server 8.1.7 database as the third tier back end storage. The Emulator Servlet was hosted by Jakarta Tomcat version 3.2.2. All machines used in our experiments ran Solaris 5.8. The application server used in this study is one of the market leaders. We are not able to release the name due to licensing restrictions. In all of our experiments, a single instance of the application server hosted the entire middle tier. Many commercial application servers, including ours, provide a clustering mechanism that links multiple server instances running on the same, or different machines. The scaling data presented here does not include any such feature and only represents the scaling of a single application server instance, running in a single JVM.

We used the Solaris tool psrset to restrict the application server threads to only run inside a subset of the available processors on the machine. The psrset mechanism also prevents other processes from running on processors within the processor set. This technique enabled us to measure the scalability of the application server and to isolate it from interference by other applications running on the host machine. We used the Solaris tool cpustat to access the built-in counters on the UltraSparc II and III processors. This tool allowed us to examine the effect of scaling on the stall rates, CPI and the fraction of L2 cache misses that are satisfied with cache-to-cache transfers, etc.

3.1 Hardware Setup

We have performed our experiments on two different hardware platforms. In the first system, the application server was run on a Sun E6000. In the second, the application server was run on a Sun E15000. We refer to these systems throughout the paper by the type of machine that ran the application server.

Hardware Setup E6000

We ran the application server and the database on two identical Sun E6000s. The E6000 is a bus-based snooping multiprocessor with 16 UltraSparc II 248 MHz processors with 1MB L2 caches, and 2GB of main memory. The UltraSparc II processors are 4-wide-in-order and block on cache misses. The Emulator and driver were each run on a 500Mhz USIIe Sun Netra. All the machines were connected by a 100-Mbit Ethernet link.

Hardware Setup E15000

We ran the entire benchmark on a Sun E15000, which is a cc-nUMA distrubuted shared memory system with 4-processor nodes. It houses 48 UltraSparc III 900MHz processors each with 8MB L2 caches and a total of 48 GB of main memory. The Driver, Emulator and Database were run on a separate OS domain from the application server. The two domains were connected by a 100-Mbit Ethernet link.

3.2 Benchmark Tuning

Tuning ECperf is complicated because there are several layers of software to configure. At the lowest level, we configured Solaris to enable Intimate Shared Memory (ISM). Enabling ISM increases the page size from 8 KB to 4 MB and allows sharing of page tables between threads. These optimizations greatly increase the reach of the TLB, which would otherwise be much smaller than the large heap of the application server. We configured the JVM by testing various thread synchronization and garbage collection settings. We found that the default thread synchronization method gave us the best throughput on ECperf when the application server was well tuned. In all cases the heap size was set to the largest value that our system could support, 1624 MB on the E6000, and 3500MB on the E15000. We tuned the garbage collection mechanism in the virtual machine by increasing the maximum sizes of the new generation to 700MB, and 1GB on the E6000 and E15000 systems respectively. Finally, we tuned the application server for each processor set size by running the benchmark repeatedly with a wide range of values for the size of the execution queue thread pool and the database connection pool**.** For each processor count, the configuration settings used were those that produced the best throughput.

3.3 Monitoring Results

Our monitoring results are based on the output data of the benchmark itself and on the sampling of the integrated event counters in the UltraSparc processors accessed through the tool cpustat. Based on the results of the benchmark and the cpustat output, we were able to measure the effect of scaling on the throughput, the CPI, and the fraction of L2 cache misses that result in a cache-to-cache transfer.

3.3.1 Throughput

Figure 3 shows the scaling of system throughput normalized to the uniprocessor throughput of the E6000 as the middle-tier of each system was scaled from 1 to 12 processors. On the E6000, ECperf scales effectively up to 10 processors and even super linearly up to 6 processors. The uniprocessor throughput for the E15K is about four times higher than the E6000. The E15K scales well up to 4 processors and has a positive, although very modest, throughput improvement to 8 processors. This phenomena can be explained by the topology of the E15K. There are 4 CPUs per board, and each board forms a snooping domain. Coherence across boards is maintained by a directory. In the 2 and 4 processor configurations, all communication occurs within a single snooping domain, and no long-latency directory operations are necessary. The 6 processor and larger configurations suffer a performance penalty because inter processor communication must span boards.

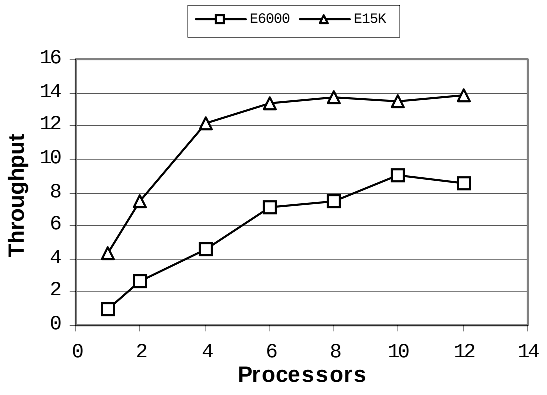

Figure 3 Throughput (Normalized to the E6000 Uniprocessor)

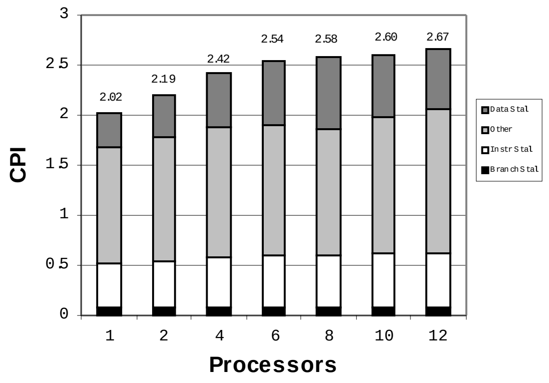

Figure 4 CPI Contribution Breakdown (E6000)

3.3.2 CPI Contributions

Figure 4 illustrates that the user-level number of Cycles-Per-Instruction (CPI) for ECperf increases steadily from 1 to 6 processors on the E6000. We were not able to provide an accurate measure of the kernel CPI since cpustat counts time in the idle loop as kernel time. The increase in user CPI is almost entirely due to an increase in the data stall rate which increases from 17% to 28% for these processor counts. Branch stall contributes about 3% to the execution time and instruction stall is also fixed at about 20%. An interesting observation is that although ECperf scales super-linearly from 1-6 processors, the average CPI is increases. The number of instructions executed per BBop decreases of that same range. We hypothesize that the drop is due to object-level caching in the application server. It is possible that the application server is able to re-use objects more efficiently if it is able to run multiple threads simultaneously.

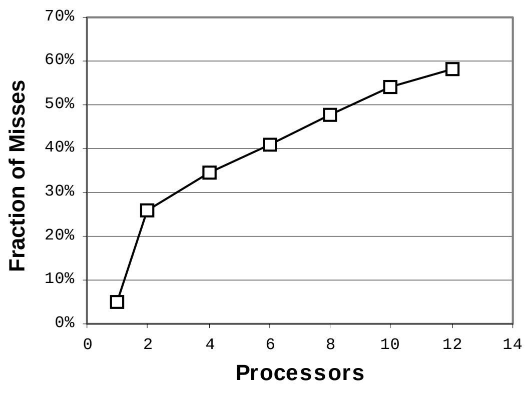

Figure 5 Cache-to-Cache Transfer Ratio (E6000)

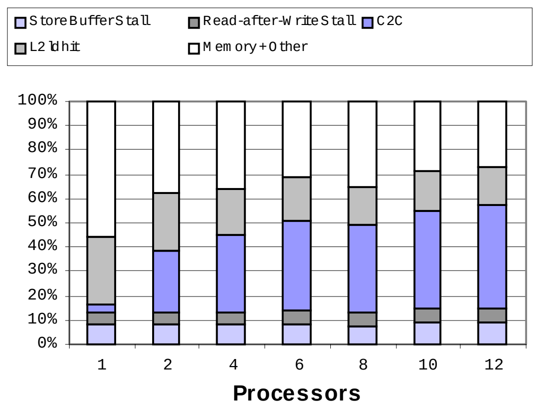

Figure 6 Estimated Data Stall Breakdown (E6000)

3.3.3 Cache-to-Cache Transfer Ratio

We found that the ratio of cache-to-cache transfers to L2 misses in ECperf is comparable to the ratio previous papers have reported for other commercial workloads, including OLTP. Figure 5 shows that the fraction of L2 cache misses that hit in another cache starts at 26% for two processors and increases rapidly to 58% for twelve processors. This ratio is important because the time it takes for a processor to satisfy an L2 miss out of another processor's cache varies widely between different memory systems. On the E6000, the latency of cache-to-cache transfer is approximately 27% longer than the latency to access to main memory [7]. For NUMA memory systems, this penalty is typically much higher, especially considering multi-hop transactions in directory-based protocols. A latency penalty in the 200-300% range is not uncommon [6]. The E15K used in our experiments, which is a fairly uniform NUMA has a cache-to-cache latency penalty of approximately 50% [4].

We can see in Figure 5 that the ratio of cache-to-cache transfers to total L2 misses increases as we add more CPUs. This result is caused by a large fraction of data accesses with a migratory sharing pattern. With only two CPUs, the likelihood of the same CPU accessing the same migratory item twice in a row is 50%, which is why we will observe about half as many cache-to-cache transfers compared with a larger number of CPUs. Figure 4 illustrates how the CPI increases with the number of CPUs and that the dominating cause for this is the increased data stall time. Figure 6 provides a breakdown of components of data stall time. In it we see that, the dominating part of the data stall (about 70 percent) is due to misses in the L2 cache. Figure 6 also shows that part of the CPI increase for higher CPU counts comes from the increased portion of L2 misses that cause cache-to-cache transfers, which take about 95 cycles to satisfy on an unloaded E6000, compared to 75 cycles to access main memory.

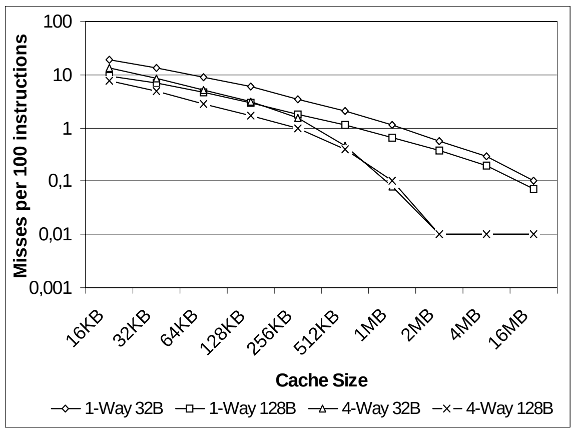

Figure 7 Cache Miss Ratio (Instructions)

4 Simulation Environment

We used the Simics full-system simulator [9] to simulate ECperf running on several different hardware configurations. Simics is an execution driven simulator that runs unmodified SPARC code and unmodified Solaris. We used the Sumo cache simulator, which attaches to Simics, to determine the cache performance of ECperf on various cache configurations. For these simulations, we violated the ECperf specifications by running the entire benchmark on a single (simulated) machine. We hope to repeat the experiments using multi-machine simulation results in the near future.

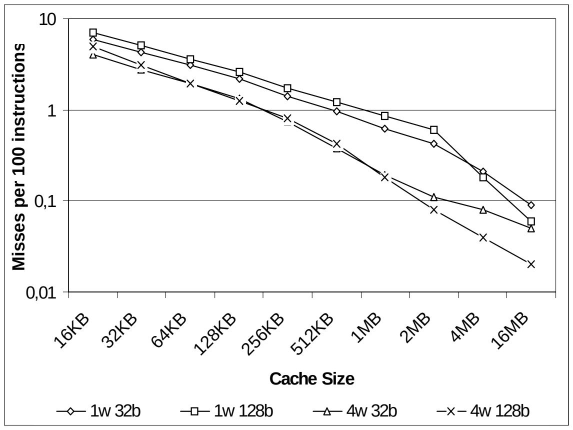

Figure 8 Cache Miss Rate (Loads)

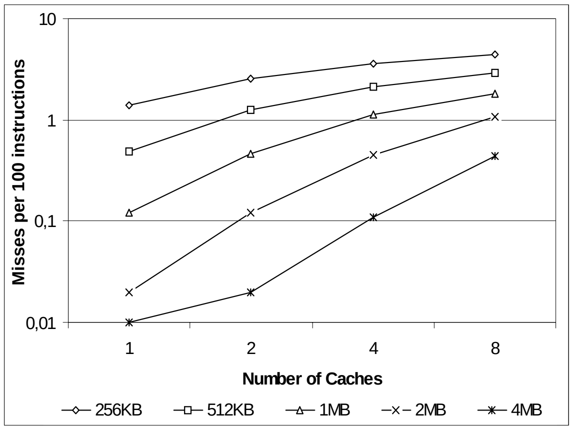

Figure 10 Instruction Misses (Aggregate Cache Size Fixed)

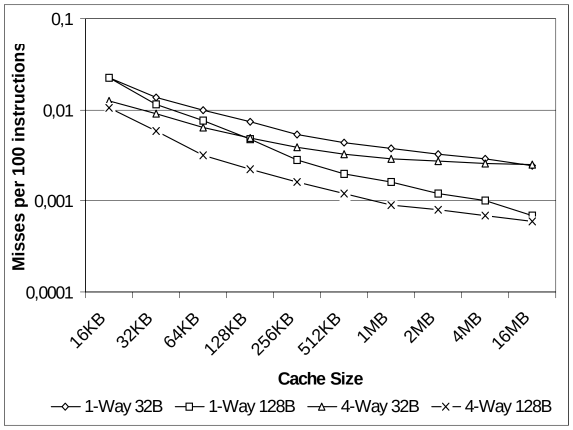

Figure 9 Cache Miss Rate (Stores)

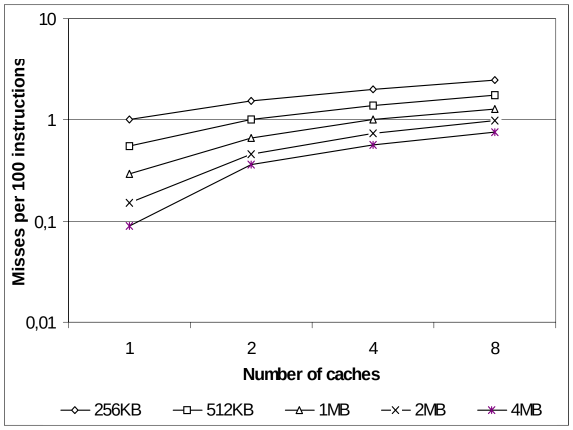

Figure 11 Load Misses (Aggregate Cache Size Fixed)

4.1 Cache Performance

To measure the size of the primary working set of the application server running ECperf, we configured Simics to model a 2-processor E6000-like SPARC V9 system with 2GB of main memory running Solaris 8. The application server was isolated from the rest of the benchmark by using psrset to restrict it to run on only one of the processors. The reported cache statistics were taken from that processor only. We varied the size, associativity and block size of the simulated cache.

As demonstrated by Figure 7, the application server has a small, well-defined instruction working set that is approximately 1-2MB. A 4MB associative cache can hold the entire application and feed the processor with nearly zero misses. Both the load and instruction miss rates are very sensitive to associativity, much more so than to line size. Figure 8 shows that the load miss rate of ECperf on a direct-mapped cache is much higher than the miss rate of a 4-way set associative cache of the same size for both caches with 32 and 128 byte cache lines. This suggests that many of the misses in ECperf are due to cache conflicts.

The cache behavior of stores is different from that of instructions and loads. Figure 8 illustrates that with 16KB caches, the miss rate is more sensitive to associativity than line size. However, as the cache size increases, the effect of the line size increases rapidly. We suspect that the abundance of spatial locality of stores is due to either or both garbage collection and object initialization in the JVM, both of which involve storing to consecutive memory addresses. Previous work has shown that the cache miss rate during garbage collection is typically much higher than during application code [13]. Since we found that the application server spent 2% to 10% of the benchmark interval performing garbage collection, it is possible that a large percentage of the store misses occur during garbage collection, however, further experiments are necessary to

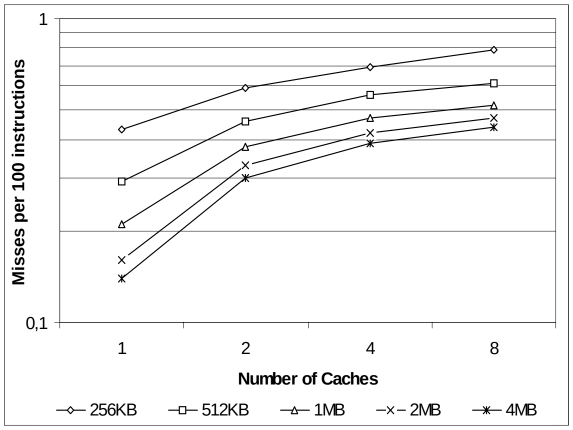

Figure 12 Store Misses (Aggregate Cache Fixed)

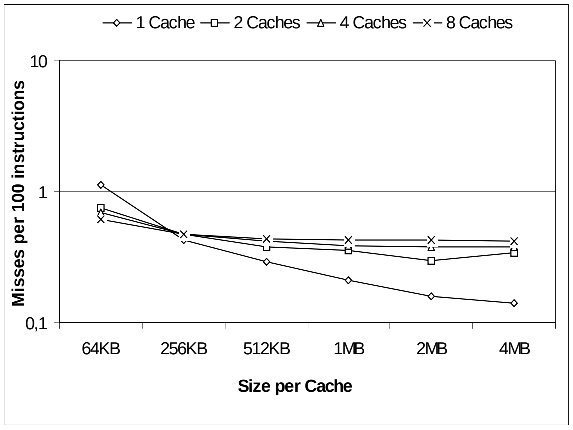

Figure 13 Store Misses (Aggregate Size Increasing)

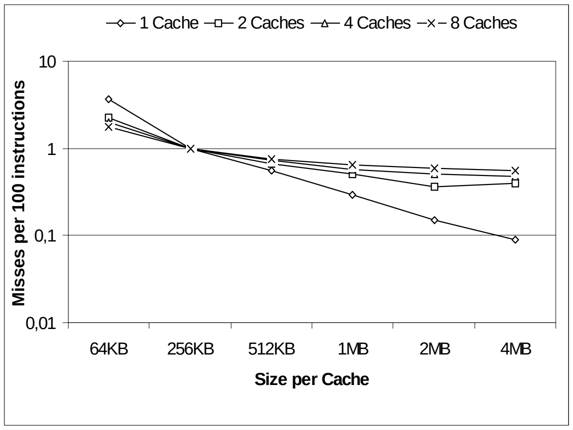

Figure 14 Load Misses (Aggregate Size Increasing)

determine if the observed spatial locality of stores in ECperf is in fact due to garbage collection.

4.2 Shared Caches

In order to evaluate the suitability of this workload to a shared-cache memory system, we also simulated a 16-processor machine where the application server was bound to 8 processors. We filtered out the memory requests from the other 8 processors, and fed only the requests from the application server processors to our memory system simulator. We then configured our simulator to simulate various degrees of sharing between the 8 processors. Figures 10-14 present the cache miss rates of our various shared-cache configurations. The x-axis for each graph is the number of separate caches in the memory system. For example, if the number of caches is 4, it indicates that two processors share each cache. Our results are based on simulations of 4-way associative caches with a cache block size of 64 bytes. We expected to see some positive effects from reducing the amount of coherence misses as well as some negative effects from cache pollution. Our results, however, clearly indicate that the cache pollution is negligible and that coherence misses are the dominating factor in the miss rate of ECperf for caches of this size and associativity. As illustrated by Figures 11 and 12, sharing has a positive effect on all cache sizes; the configurations with fewer and more widely shared caches performed better in every case. For loads and stores, an 8 way shared 1MB cache even had a lower miss rate than 8 separate 1MB caches, even though the total cache memory was only 1/8 the size. As we can see in Figures 13 and 14, with caches larger than 1MB, there is a clear incentive to share caches between processors.

Figure 10 shows that cache sharing also had a positive effect on the instruction cache. Due to the lack of coherence misses the miss rate reduction can be attributed to reducing cold instruction misses by inter-processor prefetching. ECperf is well suited to cache sharing at the second level. Caches of the size typically found in L1 caches are too small to divide effectively (see Figures 13 and 14), but 1MB and 4MB caches are more effective when shared. This suggests that ECperf is a homogenous workload and that threads across several processors are executing the same code and accessing the same data.

5 Related Work

This paper extends previous work by examining an example of an important new class of commercial applications, EJB-based middleware. Other papers have presented the behavior of commercial applications. Among the most notable are those that describe the behavior of DBMS's running the TPC benchmarks TPC-C and

TPC-H [1][2]. Ailamaki et al report that DBMS's spend much of their time handling level-1 instruction and level-2 data misses [1]. Barroso et al report that the memory system is a major factor in the performance of DBMS workloads, and that OLTP workloads are particularly sensitive to cache-to-cache transfer latency, especially in the presence of large second level caches [2]. These studies demonstrate that the execution of time of DBMS is closely tied to the performance of the memory system. The large memory buffers and instruction footprints of these applications can only be captured in large caches. Furthermore, even though large caches may capture the working sets of a DBMS, the coherence traffic will still produce many cache misses.

Other studies have also examined Java workloads. Luo and John present a characterization of VolanoMark and SPECjbb2000 [8]. These benchmarks, while both java applications and designed to represent commercial workloads, are not specifically middle-tier. VolanoMark behaves quite differently than ECperf because of the high number of threads it creates. In VolanoMark, the server creates a new thread for each client connection. The application server that we have used, in contrast, leverages the thread pooling capability of a commercial application server to share threads between client connections. As a result, the middle tier of the ECperf benchmark spends much less time in the kernel than VolanoMark. SPECJbb also has a much lower kernel component than VolanoMark, but combines the functionality of all three tiers of a 3-tiered system into a single application, making it impossible to isolate the results of the middle tier. Cain et al describe the behavior of a Java Servlet implementation of TPCW, which models an online bookstore [3]. Though the Servlets in their implementation are also middleware, that workload is also quite different than ECperf, since it does not maintain session information for client connections. There is, in fact, no communication between the Servlet threads that process the client requests. Marden et al compare the memory system behavior of a PERL CGI script and a Java Servlet [10]. Chow et al measure uniprocessor performance characteristics on transactions from the ECperf benchmark [5]. They present correlations between both the mix of transaction types and system configuration to processor performance.

6 Conclusions

This paper presents a detailed characterization of a leading commercial application server running ECperf, a benchmark that is representative of an important new class of multiprocessor server workloads. Using a combination of monitoring experiments on real machines and detailed execution-driven simulations, we have measured the scalability, cache miss rates and the effectiveness of cache-sharing for this workload. We find that the working sets of ECperf are small for a commercial workload, and that the miss rate is particularly sensitive to the associativity of both the instruction and data caches. We found that the cache miss rate of instructions and loads in a uniprocessor system fell to nearly zero for caches 4MB and larger with 4-way associativity. We observed a high number of cache-to-cache transfers in our multiprocessor configurations of ECperf. Previous studies have noted a higher rate of such cache-to-cache transfers on OLTP than on other types of workload. Barroso et al report that OLTP generated a cache-to-cache transfer ratio of 55% with 4MB L2 caches [2]. Our results indicate that the ratio of cache-to-cache transfers in ECperf is comparable to that of OLTP.
Based on our results, we conclude that a memory system that trades cache capacity for a decrease in cache-to-cache transfer latency will be well suited for ECperf. Our simulations demonstrated that ECperf is particularly well suited to a shared-cache memory system. The data miss rate of ECperf on a single 1MB cache shared 8 ways is lower than the miss rate of 8 separate 1MB caches, even though the latter has 8 times the total cache capacity.

7 Acknowledgments

We would like to thank Michael Koster and Paul Caprioli for introducing us to the ECperf benchmark and for their fine mentoring. We also sincerely thank Jim Nilsson for providing us with the Sumo cache simulator, Akara Sucharitakul for his help configuring and tuning ECperf, Dan Sorin and Alvy Lebeck for their insightful comments, and John Busch for keeping us focused throughout the project.

8 References

[1] Anastassia G. Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, September 1999.

[2] 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, June 1998.

[3] 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, January 2001.

[4] Alan Charlesworth. The Sun Fireplane Interconnect. In *Proceedings of SC2001*, November 2001.

[5] Kingsum Chow, Manesh Bhat, Ashish Jha, and Colin Cummingham. Characterization of Java Application Server Workloads. In *IEEE 4th Annual Workshop on Workload Characterization in conjuntion with MICRO-34*, pages 175–181, 2002.

[6] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Von Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[7] Erik Hagersten and Michael Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, January 1999.

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

[9] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[10] 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*, February 2002.

[11] ECperf Home Page. ECperf. http://java.sun.com/j2ee/ecperf/.

[12] SPEC. SPEC jAppServer Development Page. http://www.spec.org/osg/jAppServer/.

[13] Benjamin Zorn. The Effect of Garbage Collection on Cache Performance. Technical Report CU-CS-528-91, CUCS, May 1991.