Appears in the proceedings of the
9th Annual International Symposium on High-Performance Computer Architecture (HPCA-9)
Anaheim, CA, February 8-12, 2003

Memory System Behavior of Java-Based Middleware

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

Java-based middleware, and application servers in particular, are rapidly gaining importance as a new class of workload for commercial multiprocessor servers. SPEC has recognized this trend with its adoption of SPECjbb2000 and the new SPECjAppServer2001 (ECperf) as standard benchmarks. Middleware, by definition, connects other tiers of server software. SPECjbb is a simple benchmark that combines middleware services, a simple database server, and client drivers into a single Java program.ECperf more closely models commercial middleware by using a commercial application server and separate machines for the different tiers. Because it is a distributed benchmark, ECperf provides an opportunity for architects to isolate the behavior of middleware.

In this paper, we present a detailed characterization of the memory system behavior of ECperf and SPECjbb using both commercial server hardware and Simics full-system simulation. We find that the memory footprint and primary working sets of these workloads are small compared to other commercial workloads (e.g., on-line transaction processing), and that a large fraction of the working sets are shared between processors. We observed two key differences between ECperf and SPECjbb that highlight the importance of isolating the behavior of the middle tier. First, ECperf has a larger instruction footprint, resulting in much higher miss rates for intermediate-size instruction caches. Second, SPECjbb's data set size increases linearly as the benchmark scales up, while ECperf's remains roughly constant. This difference can lead to opposite conclusions on the design of multiprocessor memory systems, such as the utility of moderate sized (i.e., 1 MB) shared caches in a chip multiprocessor.

This work is supported in part by the National Science Foundation, with grants EIA-9971256, EIA-0205286, and CDA-9623632, the PAMP research program supported by the Swedish Foundation for Strategic Research, a Wisconsin Romnes Fellowship (Wood), and donations from Intel Corporation, IBM, and Sun Microsystems.

1. Introduction

Architects have long considered On-Line Transaction Processing (OLTP) and Decision Support Systems (DSS) as important workloads for multiprocessor servers. The recent shift toward 3-tier and N-tier computing models has created a large and rapidly-growing market for Java-based middleware, especially application servers. Still, middleware workloads are not yet well understood, and there are few accepted benchmarks that measure the performance of middle-tier applications. This is due both to the recent emergence of middleware as a mainstream workload and to the fact that 3-tier workloads are by nature difficult to install, tune and run.

We present a detailed characterization of two Java-based middleware benchmarks, SPECjbb and ECperf (now SPECjAppServer2001 [17]), running on shared-memory multiprocessors. ECperf more closely resembles commercial middleware applications because it runs on top of a commercial application server and is deployed on a 3-tiered system. The distributed nature of ECperf also facilitates monitoring the behavior of each tier independently. ECperf, however, is difficult to install and run. It requires the coordination of several machines and several pieces of software. SPECjbb is also a Java middleware benchmark. It is an attractive alternative to ECperf because although it models a 3-tiered system, it is a single Java program that can be run on any Java Virtual Machine (JVM). SPECjbb includes many common features of 3-tiered systems in a single program running on a single machine.

The goal of this paper is to understand the memory system behavior of these middleware benchmarks, to gain insight into the behavior of Java-based middleware, and to provide useful data and analysis to memory systems designers targeting middle-tier servers. We focus on mid-range (up to 16 processor) shared-memory multiprocessors because many application servers target these systems. We also investigate whether or not the simple SPECjbb benchmark behaves similarly enough to the more complex ECperf to be considered representative of commercial middleware applications.

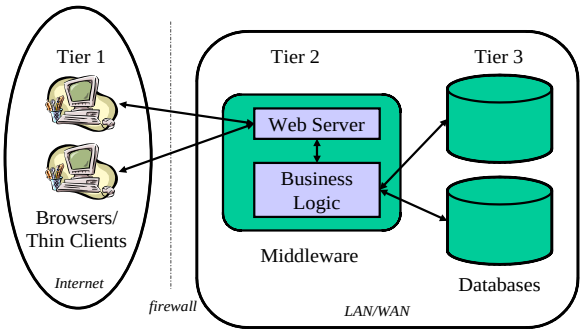

Figure 1: 3-Tiered Systems

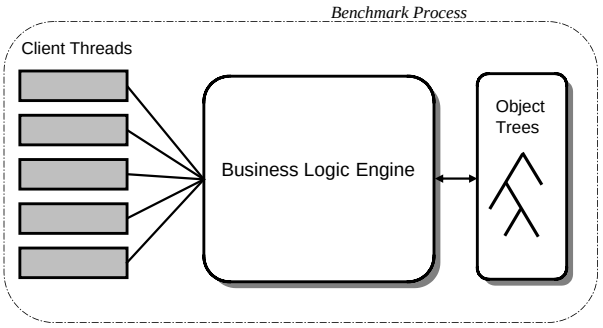

Figure 2: SPECjbb Overview

We find that these Java-based middleware applications have moderate CPIs compared to previously-published commercial workloads (between 2.0 and 2.8 for ECperf). In particular, memory related stalls are low, with misses to main memory accounting for as little as 15% of the data stall time and 5% of total execution time. Conversely, sharing misses occur frequently in both workloads, accounting for over 60% of second-level cache misses on larger systems. SPECjbb is similar to ECperf in many ways, but there are important differences between the two benchmarks. ECperf has a larger instruction working set, but a lower data cache miss rate. Furthermore, the memory footprint of ECperf remains nearly constant as the benchmark scales up, whereas the memory use of SPECjbb grows linearly with database size. We show that this difference can lead to opposite conclusions on some design decisions, like the utility of shared level-two caches in a chip multiprocessor.

2. Background

The emergence of the Internet and World Wide Web has triggered a shift in enterprise computing from a two-tiered, client-server architecture to a 3-tiered architecture (see Figure 1), where a Web browser is now used universally as a database client. For databases, connection to the Web allows users to access data without installing a client program. For Web pages, databases provide dynamic content and permanent storage. Software that connects databases to Web pages is known as "middleware." Much of the middleware used today is written in Java. Two of the most popular Java middleware architectures are Java Servlets and Enterprise Java Beans (EJB). The two are often used together, with Servlets implementing the presentation logic and EJB providing the business rules. Application servers host both Servlets and EJB and provide them with communication with both back-end databases and front-end web clients.

Recently, Web-connected database applications have also 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. SPECjbb Overview

SPECjbb is a software benchmark designed to measure a system's ability to run Java server applications. Inspired by the On-Line Transaction Processing Benchmark TPC-C, SPECjbb models a wholesale company with a variable number of warehouses. Beyond the nomenclature and business model, however, there are few similarities between TPC-C and SPECjbb. TPC-C is intended to measure the performance of large-scale transaction processing systems, particularly databases. In contrast, SPECjbb was written to test the scalability and performance of JVMs and multiprocessor servers that run Java-based middleware. It emphasizes the middle-tier business logic that connects a back-end data store to a set of thin clients, and is implemented entirely in Java.

SPECjbb models a 3-tiered system, but to make the benchmark portable and easy to run, it combines the behavior of all 3 tiers into a single application (see Figure 2). Instead of using a commercial database engine like most real 3-tiered systems, SPECjbb stores its data in memory as trees of Java objects [18].

The SPECjbb specification calls for running the benchmark with a range of warehouse values. In an official SPECjbb run, the benchmark is run repeatedly with an increasing number of warehouses until a maximum throughput is reached. The benchmark is then run the same number of times with warehouse values starting at the maximum and increasing to twice that value. Therefore, if the best throughput for a system comes with n warehouses, 2n runs are made. The benchmark score is the average of runs from n to 2n warehouses. This large number of separate benchmark runs would take prohibitively long in simulation. Therefore, in our simulation experiments, we selected 3 values for the number or warehouses to represent the range of values that would be included in a publishable SPECjbb result for our hardware configuration. In order to simply our monitoring simulations, we report results from the steady state inter-

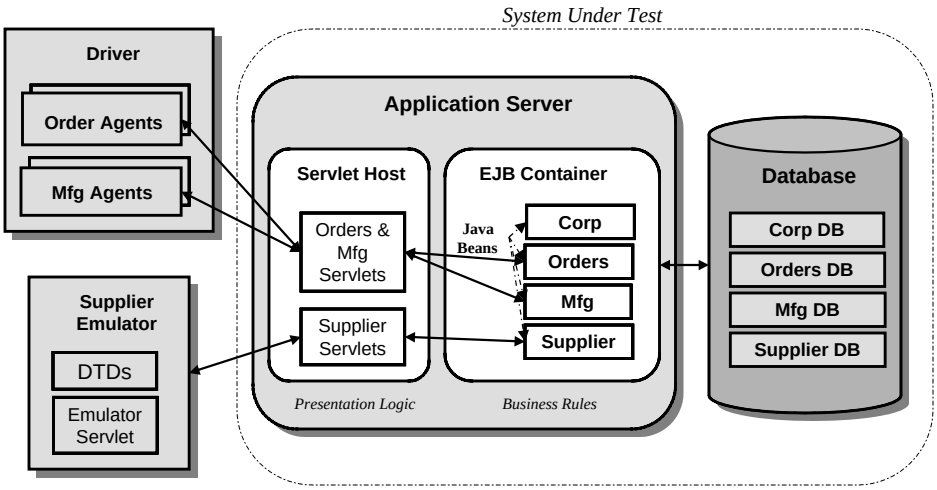

Figure 3: ECperf Setup

val of SPECjbb running with the optimal number of warehouses at each system size.

2.2. ECperf Overview

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. The presentation layer is implemented with Java Servlets, and the business rules are built with EJB. The application is divided into the following four domains, which manage separate data and employ different business rules. The Customer Domain models the actions of customers who create, change and inquire about the status of orders. The customer interactions are similar to On-Line Transaction Processing (OLTP) transactions. 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. The Supplier Domain models interactions with external suppliers. The parts inventory is updated as purchase orders are filled. Finally, the Corporate Domain tracks customer, supplier and parts information.

The ECperf specification supplies the EJB components 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.

Four separate agents participate in the ECperf benchmark, each of which is run on a separate machine or group of machines. Each of these parts is represented by a box in Figure 3.

Application Server The application server, shown in the center of Figure 3, hosts the ECperf Java Beans. Together, they form the middle tier of the system, which is the most important component to performance on ECperf.

Database 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 provide atomic transactions.

Supplier Emulator Suppliers are emulated by a collection of Java Servlets hosted in a separate web container.

Driver The driver is a Java program that spawns several threads that model customers and manufacturers.

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 "Benchmark Business Operation," or "BBop." Performance on ECperf is measured in terms of BBops/minute. Although performance on ECperf is measured in terms of throughput, the benchmark specification requires that 90% of all transactions are completed within a fixed time [9, 15]. In our experiments, however, we relaxed the response time requirement of ECperf and tuned our system to provide the maximum throughput regardless of response time.

2.3. Enterprise Java Beans

ECperf is implemented using Enterprise Java Beans (EJB), a part of the Java 2 Enterprise Edition (J2EE) standard. EJB are reusable Java components for server-side applications. In other words, they are building blocks for web-service applications. They are not useful until they

are deployed on an 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.

2.4. Java Servlets

Servlets are Java classes that run inside a dynamic web server. Servlets can communicate with a back-end database through the Java DataBase Connectivity (JDBC) API. Session information can be passed to Servlets either through browser cookies or URL renaming.

2.5. 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 as a host for presentation logic, including Java Servlets. As an EJB container, it provides required services such as database connections and persistence management. It also provides better performance and scalability than a naïve implementation of the J2EE standard.

Three important performance features of our particular server are thread pooling, database connection pooling, and object-level caching. 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, rather than creating new ones and later 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 in the Operating System that database connection pooling does in 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. Object-level caching increases performance in the application server because instances of components (beans) are cached in memory, thereby reducing database queries and memory allocations.

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 in section 4 does not include this feature and only represents the scaling of a single application server instance, running in a single JVM.

3. Methodology

We used a combination of monitoring experiments on real hardware and detailed full-system simulation to measure the memory system behavior of our middleware workloads. The native hardware enabled us to perform our measurements on a complete run of the benchmarks while our simulation study offered us the opportunity to change the memory system parameters. On the native hardware, we used the Solaris tool *psrset* to restrict the application threads to only run on a subset of the processors available 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 applications and to isolate them from interference by other applications running on the host machine.

3.1. Hardware Setup

We ran both SPECjbb and the application server of ECperf on a Sun Enterprise 6000 server. The E6000 is a bus-based snooping multiprocessor with 16 248-MHz UltraSPARC II processors with 1 MB L2 caches and 2 GB of main memory. The UltraSPARC II processors are 4-wide and in-order issue. For ECperf, we ran the database on an identical Sun E6000, and the supplier emulator and driver were each run on a 500 MHz UltraSPARC IIe Sun Netra. All the machines were connected by a 100-Mbit Ethernet link.

3.2. Benchmark Tuning

Tuning Java server workloads is a complicated process because there are several layers of software to configure, including the operating system, the JVM, and the application itself. Tuning 3-Tier Java applications is more complicated still, because the application server and database must be properly configured as well.

Operating System (Solaris 8) We optimized Solaris for running large server programs by enabling Intimate Shared Memory (ISM), which increases the page size from 8 KB to 4 MB and allows sharing of page table entries between threads. This optimization greatly increases the TLB reach, which would otherwise be much smaller than the application server's large heap.

JVM (HotSpot 1.3.1) 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 and

SPECjbb. In all cases, the heap size was set to the largest value that our system could support, 1424 MB. We tuned the garbage collection mechanism in the virtual machine by increasing the size of the new generation to 400 MB. A large new generation leads to fewer, but longer, partial collections and better total throughput. Our multiprocessor simulations of SPECjbb were run with HotSpot 1.4.0. In order to be as consistent as possible with both our uniprocessor simulations and the multiprocessor simulations of ECPerf, we used the same heap and new generation sizes in all of our experiments.

Application Server For ECperf, 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.

Database ECperf uses a small database, which fit entirely in the buffer pool of our database server. We found that the performance of ECperf was unaffected by other database settings.

3.3. Simulation Environment

We used the Simics full-system simulator [11] to simulate ECperf and SPECjbb running on several different system configurations. Simics is an execution-driven simulator that models a SPARC V9 system accurately enough to run unmodified Solaris 8. To determine the cache behavior of the applications without communication, we configured Simics to model a 1-processor E6000-like SPARC V9 system with 2GB of main memory running Solaris 8. To run ECperf, we simulated four such machines connected by a simulated 100-Mbit Ethernet link. The reported cache statistics for ECperf were taken from the simulated machine that ran the application server.

For these experiments we extended Simics with a detailed memory system simulator [13]. The memory system simulator allowed us to measure several cache performance statistics on a variety of caches with different sizes, associativities and block sizes. In order to evaluate the communication behavior of these workloads and their suitability to a shared-cache memory system, we also simulated multiprocessor configurations of each workload. We were not able to simulate a multi-tiered configuration of ECperf running on a multiprocessor. Instead, we simulated a single 16-processor machine where the application server was bound to 8 processors. We then 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 use the methodology proposed by Alameldeen, et al. [2] to account for the inherent variability of multi-threaded commercial workloads. We present the means and standard deviations (shown as error bars) for all measured and most simulated results.

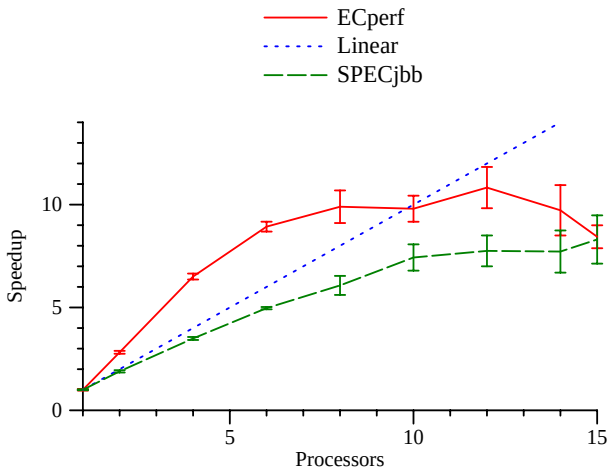

Figure 4: Throughput Scaling on a Sun E6000

4. Scaling Results

Java-based middleware applications, like most commercial workloads, are throughput-oriented. Understanding how these applications scale up to both larger multiprocessors and larger data sets is important for both hardware and software developers. In this section, we analyze how ECperf and SPECjbb scale on a Sun E6000 system.

Despite our best efforts to tune these workloads, we were unable to even come close to achieving linear speedup. Figure 4 shows that ECperf scales super-linearly from 1 to 8 processors, but scales poorly beyond 12 processors. ECperf achieves a peak speedup of approximately 10 on 12 processors, then performance degrades for larger systems. SPECjbb scales up more gradually, leveling off after achieving a speedup of 7 on 10 processors.

In the remainder of this section we present an analysis of the factors that contribute to the limitations on scaling. We find that both benchmarks experience significant idle time (approximately 25%) for systems with 10 or more processors, apparently due to contention for shared software resources. Memory system stalls are the second major factor, causing the average cycles per instruction to increase by as much as 40%. Finally, although garbage collection does impact performance, on larger systems it accounts for only a fraction of the difference between measured and linear speedup.

4.1. Resource Contention

We used a variety of Solaris measurement tools to identify the bottlenecks in ECperf and SPECjbb. Figure 5 shows a breakdown of the time spent in various execution modes as measured by the Solaris tool *mpstat*. The four modes are running the operating system (system), run-

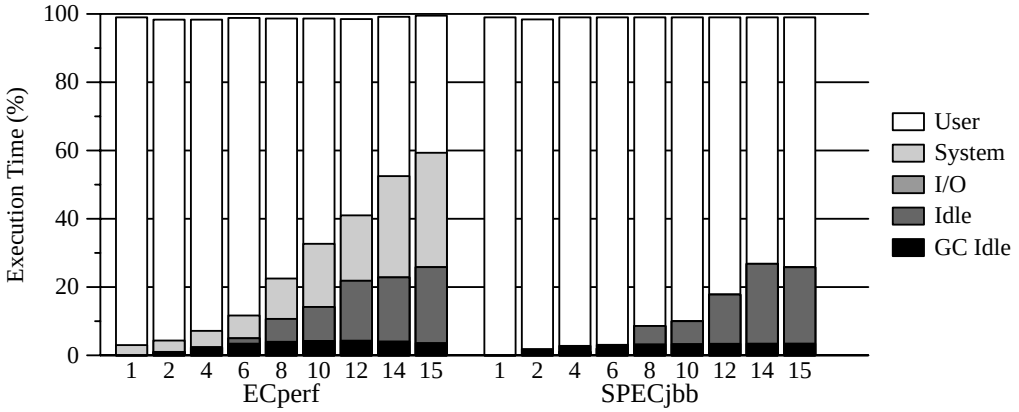

Figure 5: Execution Mode Breakdown vs. Number of Processors

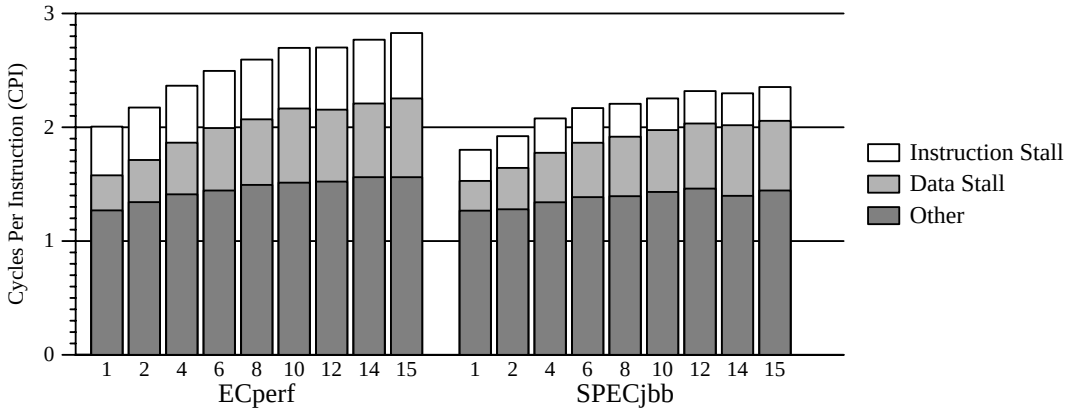

Figure 6: CPI Breakdown vs. Number of Processors

ning the benchmark (user), stalled for I/O (I/O), and stalled for other reasons (idle).

Figure 5 illustrates one important difference between ECperf and SPECjbb. ECperf spends significant time in the operating system, while SPECjbb spends essentially none. This is not surprising, since SPECjbb emulates all three tiers on a single machine, using memory-based communication within a single JVM. Conversely, ECperf uses separate machines for each tier, requiring communication via operating system-based networking code. For ECperf, the system time increase from less than 5% for a single-processor run, to nearly 30% for a 15-processor system. We hypothesize, but have been unable to confirm, that the increase in system time arises from contention in the networking code.

Both workloads incur significant idle time for larger system sizes, reaching 25% for 15 processors. Some of this idle time is due to garbage collection. Like most currently available systems, the JVM we ran uses a single-threaded garbage collector. That is, during collection only 1 processor is active and all others wait idle. We estimated the fraction of idle time due to garbage collection by multiplying the fraction of processors that are idle during collection by the fraction of time spent performing garbage collection. Figure 5 shows that the bulk of the idle time is due to factors other than garbage collection.

The increase in idle time with system size suggests that there is contention for shared resources in these benchmarks. The application server in ECperf shares its database connection pool between its many threads, and the object trees in SPECjbb are protected by locks, both of which could lead to contention in larger systems. However, the fact that the idle time increases similarly for both benchmarks indicates that the contention could be within the JVM.

4.2. Execution Time Breakdown

Idle time alone explains at most half the degradation in speedup (75% non-idle time times 15 processors is approximately 11, not the 8 we observe). To identify other limits to scalability, we used the integrated counters on the UltraSPARC II processors to measure and breakdown the average cycles per instruction (CPI) across a range of system sizes. While CPI is not a good indicator

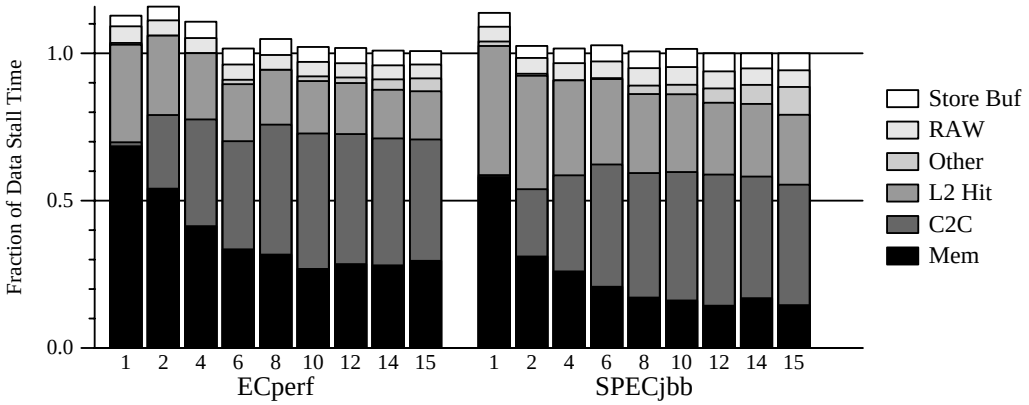

Figure 7: Data Stall Time Breakdown vs. Number of Processors

of overall performance on multiprocessors—e.g., because of the effect of the idle loop—it gives a useful indication of where the time goes.

Figure 6 presents the CPI, broken down into instruction stalls, data stalls, and other (which includes instruction execution and all non-memory-system stalls). The overall CPI ranges from 1.8 to 2.4 for SPECjbb and 2.0 to 2.8 for ECperf. These are moderate CPIs for commercial workloads running on in-order processors. Barroso, et al. report CPIs for Alpha 4100 systems of 1.3 to 1.9 for decision support database workloads and as high as 7 for a TPC-B on-line transaction processing workload. The CPI increases by roughly 40% and 33% for ECperf and SPECjbb, respectively, as the number of processors increase from 1 to 15. Assuming instruction path lengths remain constant (see Section 4.4.), the increase in CPI would account for most of the remaining performance degradation.

Figure 6 also shows that data stall time is the main contributor to the increase in CPI. On a single processor run, data stall time accounts for only 15% and 12% for ECperf and SPECjbb, respectively. However for a 15-processor system, this increases to 35% and 25% for ECperf and SPECjbb, respectively.

Figure 7 presents an approximate decomposition of the data stall time. Because some factors are estimated using frequency counts multiplied by published access times, the total does not always exactly sum to one. Approximately 60% of the data stall time is due to misses in the L2 cache, with the bulk of the remainder being L2 hits. Conversely, store buffer stalls, the cycles spent waiting for a full store buffer to be flushed, account for only 1% to 2% of the total execution time. Similarly, read-after-write hazard stalls, which occur if a load is not separated enough from a store, account for only 1% of the time.

4.3. Cache-to-Cache Transfer Ratio

Figure 7 also illustrates that cache-to-cache transfers represent a significant fraction of the data stall time for multiprocessor systems. For larger multiprocessors, cache-to-cache transfers account for nearly 50% of the total data stall time. Cache-to-cache transfers are an important factor because many multiprocessor systems take longer to satisfy a miss from a processor's cache than from main memory. On the E6000, the latency of a cache-to-cache transfer is approximately 40% longer than the latency of an access to main memory [8]. For NUMA memory systems, this penalty is typically much higher—200-300% is not uncommon [7]—because of the indirection required by directory-based protocols.

To dig deeper, we measured the cache-to-cache transfer ratio for SPECjbb and ECperf by counting the "snoop copyback" events reported in *cpustat*. In the UltraSPARC II processor, a snoop copyback event signifies that a processor has copied a cache line back to the memory bus in response to a request by another processor.

Figure 8 shows that the fraction of L2 cache misses that hit in another cache starts at 25% for two processors

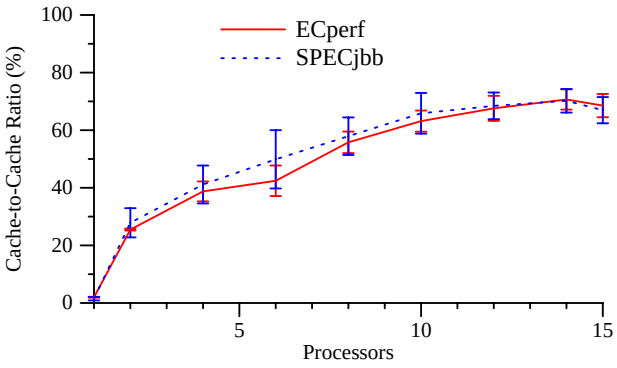

Figure 8: Cache-to-Cache Transfer Ratio

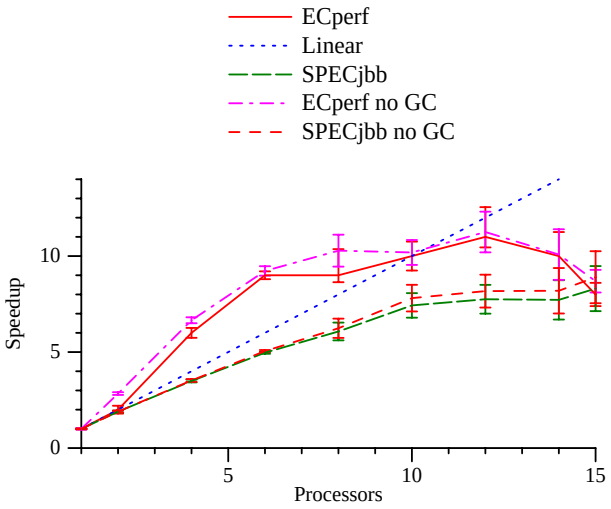

Figure 9: Effect of Garbage Collection on Throughput Scaling

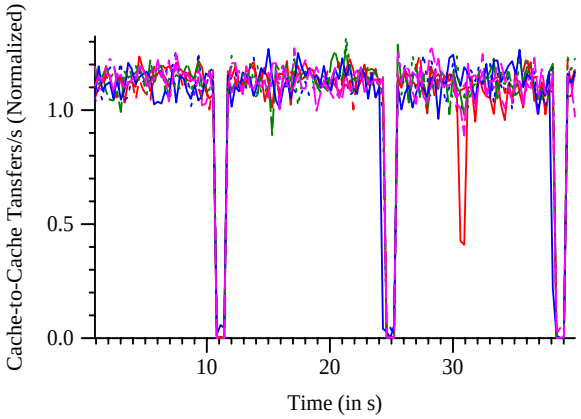

Figure 10: Cache-to-Cache Transfers Per Processor Per Second Over Time (Normalized)

and increases rapidly to over 60% for fourteen processors. This is comparable to the highest ratios previously published for other commercial workloads [3].

Figure 8 also shows cache-to-cache transfers occur even for 1 processor. These transfers are possible because the operating system runs on all 16 processors, even when the application is restricted to a single processor. Snoop copybacks occur when the processor running the benchmark responds to a request from another processor running in the operating system.

4.4. Path Length

Comparing Figure 4 to Figure 6 reveals an apparent contradiction. ECperf scales super-linearly as the system size increases from 1 to 8 processors, even though the average CPI increases over the same range. This surprising result occurs because the instructions executed per BBop decreases even more dramatically over the same range (not shown). The decrease in instruction count more than compensates for the longer average execution time per instruction. We hypothesize that this drop is due to object-level caching in the application server. Constructive interference in the object cache allows one thread to re-use objects fetched by another thread.

4.5. Garbage Collection Effects

Both workloads spend a considerable amount of time doing garbage collection. To determine the impact of the collection time on scalability, we compared the measured speedup to the speedup with the garbage collection time factored out. That is, we subtracted the garbage collection time from the runtime of the benchmark and calculated speedup in the usual way. The solid lines in Figure 9 represent the speedup of ECperf and SPECjbb as measured. The dotted lines display the speedup of the benchmarks with the garbage collection time factored out. The difference in throughput with and without the garbage collection is small, but statistically significant for ECperf up to 6 processors. For SPECjbb and ECperf on larger systems, the difference is not statistically significant.

We originally hypothesized that the high percentage of cache-to-cache transfers we observed in both SPECjbb and ECperf was due to garbage collection. Our JVM (HotSpot 1.3.1) uses a generational copying collector and is single-threaded. Therefore, during collection, all live new generation objects are copied by the collection thread regardless of which thread had created them and regardless of their location in the cache of a particular processor. For example, in a system that uses a simple MSI invalidation protocol, any new generation data in the M state cached at a processor that is not performing the collection will be read by the collector thread through a costly cache-to-cache transfer. This will result in the original copy of the data being invalidated. After the garbage collection is performed, the previous owner of the block will have to reacquire the block to access it. If the data is still residing in the garbage collector's cache, that access will result in another costly cache-to-cache transfer.

Contrary to our hypothesis, the benchmark generates almost no cache-to-cache transfers during garbage collection. We counted the number of snoop copyback events every 100 ms during a run of SPECjbb. Figure 10 illustrates this dramatic drop in the cache-to-cache transfer rate during the 3 garbage collections that occurred in our measurement interval. The HotSpot 1.3.1 JVM has an option to trace the garbage collection in a program. We used that output to verify that the decreases in the snoop copyback rate occurred during garbage collection periods. Since our JVM uses a single-threaded garbage collector, only one processor is active during the collection. That by itself would explain a drop, but Figure 10 shows that the cache-to-cache transfer rate drops to almost zero during the garbage collection periods. Even the single

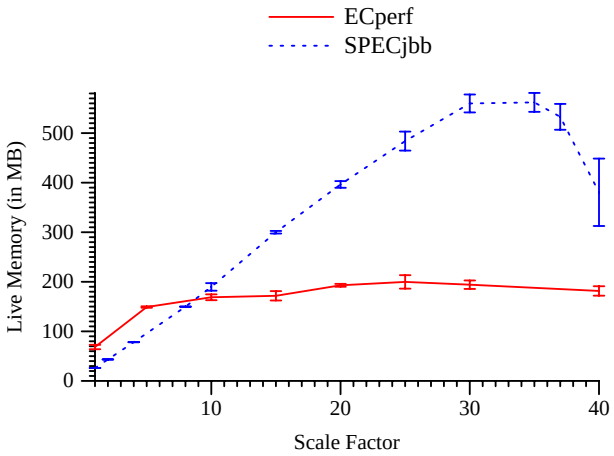

Figure 11: Memory Use vs. Scale Factor

processor which is performing the collection causes fewer cache-to-cache transfers.

4.6. Benchmark Scaling Differences

One of the most striking differences between SPECjbb and ECperf is the effect that scaling the benchmark size has on memory behavior. Like most commercial workload benchmarks, official measurements of SPECjbb and ECperf require that the benchmark size increase with input rate. In other words, faster systems must access larger databases. In SPECjbb, the input rate is set by the number of warehouses, which determines the number of threads in the program in addition to the size of the emulated database. ECperf has a similar scaling factor, the Orders Injection Rate. However, because the database and client drivers run on different machines, increasing the Orders Injection Rate has much less impact on the middle-tier memory behavior.

Figure 11 shows the average heap size immediately after garbage collection in SPECjbb and ECperf. The size of the heap after collection is an approximation of the amount of live data. As the scale factor (i.e., warehouses) increases, SPECjbb's memory use increases linearly through approximately 30. Beyond 30 warehouses, the average live memory decreases because the generational garbage collector begins compacting the older generations. This slower collection process results in dramatic performance degradation (not shown). By contrast, the memory use of ECperf increases up to an Orders Injection Rate of approximately 6, then remains roughly constant through 40. This result suggests that using SPECjbb could lead memory system designers to overestimate the memory footprints of middleware applications on larger systems.

5. Cache Performance

The previous section showed that memory system stalls were a significant detriment to scalability on the

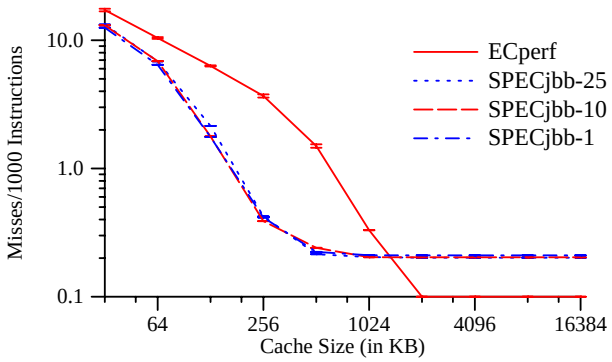

Figure 12: Instruction Cache Miss Rate

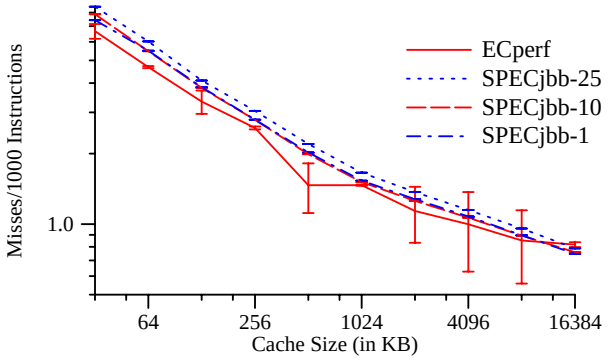

Figure 13: Data Cache Miss Rate

Sun E6000. To understand this behavior more deeply, we used full-system simulation to evaluate a variety of memory system configurations. Our simulation results show that whereas the scaling properties of these workloads are similar, the cache behavior of ECperf is quite different from that of SPECjbb. ECperf has a small data set and a low data-cache miss rate. SPECjbb puts significantly more pressure on the data cache, particularly when it is configured with a large number of warehouses. Although ECperf has a smaller data cache miss rate than even the smallest configuration of SPECjbb, a higher fraction of its total memory is shared between threads. Wider sharing and a smaller data working set make shared-caches a more effective design for that workload

5.1. Cache Miss Rates

Figure 12 and Figure 13 present the instruction and data cache miss rates, respectively, for a uniprocessor system with a range of cache sizes. All configurations assume split instruction and data caches, 4-way set associativity and 64-byte blocks. We simulated SPECjbb with three different scaling factors (1, 10, and 25 warehouses) to examine the impact of the larger memory sizes discussed in Section 4.6.

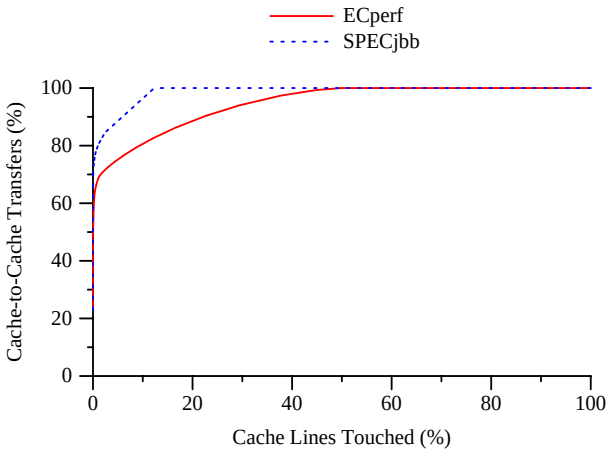

Figure 14: Distribution of Cache-to-Cache Transfers (64 Byte Cache Lines)

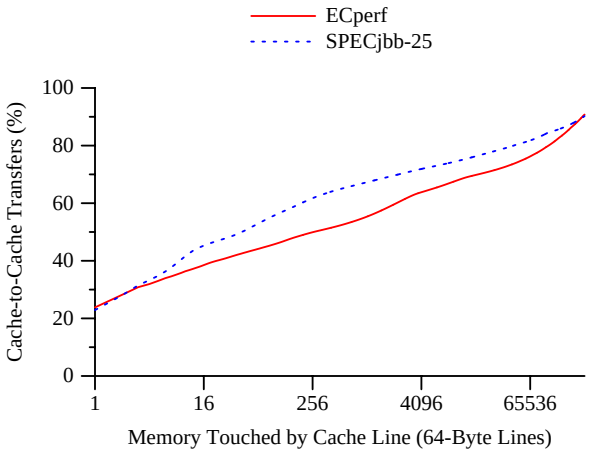

Figure 15: Distribution of Cache-to-Cache Transfers

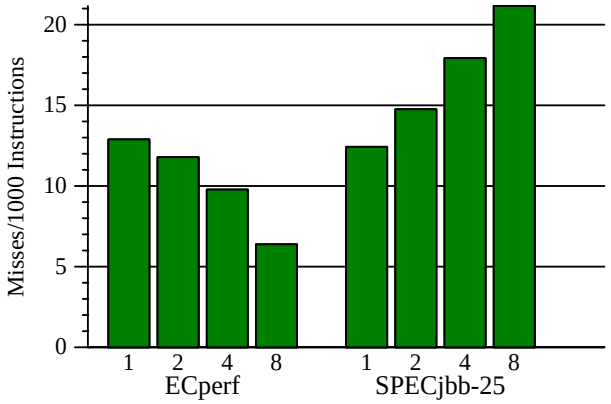

Figure 16: Cache Miss Rate on Shared Caches (Processors Per Shared 1 MB Cache)

These graphs demonstrate that both benchmarks place at most moderate demand on typical level one (L1) and level two (L2) caches. Typical L1 caches, falling in the 16 KB–64 KB range, exhibit miss rates of 10–40 misses per 1000 instructions. For typical L2 cache sizes of 1 MB and larger, the data miss rate falls to less than two misses per 1000 instructions. Instruction misses are even lower, falling well below one miss per 1000 instructions. The two benchmarks behave similarly, but do have two notable differences. First, ECperf has a much higher instruction cache miss rate for intermediate size caches (e.g., 256 KB). Second, the data miss rate for SPECjbb with one warehouse is roughly comparable to that for ECperf, but it increases by as much as 30% as the data set scales to 25 warehouses. This result is not surprising, given that SPECjbb's live data increases linearly with the number of warehouses (see Figure 11).

5.2. Communication Footprint

To provide insight into the communication behavior of the workloads, we measured the footprint of the data causing cache-to-cache transfers. As shown in Figure 14, all of the cache-to-cache transfers observed in SPECjbb came from 12% of the cache lines touched during the measurement period, and over 70% came from the most active 0.1% of cache lines. For both benchmarks, a significant fraction of the communication is likely due to a few highly contended locks. The single cache line with the highest fraction of the cache-to-cache transfers accounted for 20% of the total for SPECjbb and 14% for ECperf. This resembles earlier findings for databases and OLTP workloads [7]. In contrast to SPECjbb, however, the most active 0.1% of cache lines in ECperf account for only 56% of the cache-to-cache transfers. Furthermore, the cache-to-cache transfers are spread over half of the touched cache lines. A major contributor to this difference between ECperf and SPECjbb is SPECjbb's emulated database. The object trees that represent the database are updated sparsely enough that they rarely result in cache-to-cache transfers. Figure 15 shows the cumulative distribution of cache-to-cache transfers versus the amount of data transferred (on a semi-log plot). This graph shows that even though SPECjbb has a larger total data set, ECperf has a larger communication footprint on an absolute, not just a percentage, basis.

5.3. Shared Caches

The high cache-to-cache transfer rates of these workloads suggest that they might benefit from a shared-cache memory system, which have become increasingly common with the emergence of chip multiprocessors (CMPs) [14]. Shared caches have two benefits. First, they eliminate coherence misses between the processors sharing the same cache (private L1 caches will still cause coherence misses, but these can be satisfied on-chip much faster than conventional off-chip coherence misses). Second, compulsory misses may be reduced through inter-processor prefetching (i.e., constructive interference). The obvi-

ous disadvantage of shared caches is the potential increase in conflict and capacity misses.

To evaluate shared caches, we used Simics to model an 8-processor SPARC V9 system with four different memory hierarchies. In the base case, each processor has a private 1 MB L2 cache, for a total of 8 caches. In the other three cases, the eight processors share one, two, and four 1 MB caches. The total size of all caches is the product of the cache size (i.e., 1 MB) and the number of caches.

Figure 16 shows the data miss rates for ECperf and SPECjbb as the number of processors per cache increases. For ECperf, the benefit of reducing coherence misses more than makes up for the additional capacity and conflict misses. ECperf has the lowest data miss rate when all eight processors share a single cache, even though the aggregate cache size is 1/8 the size in the base case (i.e., private caches). Sharing had the opposite effect on SPECjbb. Even though SPECjbb had a significant fraction of cache-to-cache transfers, the larger data set size (due to the emulated database) results in an increase in overall miss rate for 1 MB shared L2 caches.

6. Related Work

This paper extends previous work by examining examples of an important emerging class of commercial applications, Java-based middleware. Cain, et al. describe the behavior of a Java Servlet implementation of TPC-W, which models an online bookstore [5]. Though the Servlets in their implementation are also Java-based middleware, that workload is also quite different than ECperf, since it does not maintain session information for client connections in the middle tier. The Servlets share a pool of database connections in that implementation like the application server in ECperf. However, no application data is exchanged between the Servlets.

Previous papers have presented the behavior of commercial applications. Among the most notable are those that describe the behavior of Database Management Systems (DBMS) running the TPC benchmarks, TPC-C and TPC-H [1][3]. 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 [3]. These studies demonstrate that the execution time of DBMS is closely tied to the performance of the memory system.

Other studies have also examined Java workloads. Luo and John present a characterization of VolanoMark and SPECjbb2000 [10]. VolanoMark behaves quite differently than ECperf or SPECjbb 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, shares 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. Marden, et al. compare the memory system behavior of a PERL CGI script and a Java Servlet [12]. Chow, et al. measure uniprocessor performance characteristics on transactions from the ECperf benchmark [6]. They present correlations between both the mix of transaction types and system configuration to processor performance. Shuf, et al. measure the cache performance of java benchmarks, including pBOB (now SPECjbb). They find that even fairly large L2 caches do not significantly improve memory system performance [16]. Their measurements, however, are limited to direct-mapped caches and uniprocessors, while we consider multiprocessors with 4-way set-associative caches. They also find that TLB misses are a major performance issue. Although we did not specifically measure TLB miss rates, we found that using the intimate shared memory (ISM) feature of Solaris, which increases the page size from 8 KB to 4 MB, increased performance of ECperf by more than 10%.

Barroso, et al. [4] and Olukotun, et al. [14] discuss the performance benefits of chip multiprocessors using shared caches. We extend their work by evaluating the impact of shared caches on SPECjbb and ECperf.

7. Conclusions

In this paper, we have presented a detailed characterization of two popular Java-based middleware benchmarks. ECperf is a complex, multi-tier benchmark that requires multiple machines, a commercial database system, and a commercial application server. In contrast, SPECjbb is a single application that is trivial to install and run. The distributed nature of ECperf makes the installation and management of that benchmark more difficult, but it also provides an opportunity to isolate the behavior of each tier individually.

We find that both workloads have low CPIs and low memory stall times compared to other important commercial server applications (e.g., OLTP). Running on the moderate size multiprocessors in our study, both workloads maintained small data working sets that fit well in the 1 MB second-level caches of our UltraSPARC II processors. More than half of all second-level cache misses on our larger systems hit in the cache of another processor.

SPECjbb closely approximates the memory behavior of ECperf except for two important differences. First, the instruction working set of SPECjbb is much smaller than that of ECperf. Second, the data memory footprint of SPECjbb is larger than that of ECperf, especially as the benchmark scales up for larger system sizes.

The difference in behavior could lead memory system designers toward different conclusions. For example,

our simulation results demonstrate that ECperf is particularly well suited to a shared-cache memory system even when the total cache size is limited to 1 MB. In contrast, the reduction in total cache capacity causes SPECjbb's performance to degrade.

This study compares two middleware benchmarks, running on a specific combination of hardware, operating system, Java virtual machine, application server, and database system. Further study is needed to determine how well these results apply to other Java middleware and different versions of the underlying hardware and software. However, we believe that as middleware becomes better understood, it will prove increasingly important to isolate its behavior from the effects of other software layers.

8. Acknowledgments

We would like to thank Paul Caprioli and Michael Koster for introducing us to the ECperf benchmark. We also sincerely thank Jim Nilsson for providing us with the Sumo cache simulator, Akara Sucharitakul for his help configuring and tuning ECperf, as well as Alaa Alameldeen, Dan Sorin and Alvy Lebeck for their insightful comments.

9. 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] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, February 2003.

[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, June 1998.

[4] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[5] 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.

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

[7] 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*, pages 13–24, November 2000.

[8] 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.

[9] Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory Characterization of the ECperf Benchmark. In *Second Annual Workshop on Memory Performance Issues (WMPI), in conjunction with ISCA-29*, 2002.

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

[11] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[12] 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.

[13] Milo M. K. Martin, Daniel J. Sorin, Anastassia Ailamaki, Alaa R. Alameldeen, Ross M. Dickson, Carl J. Mauer, Kevin E. Moore, Manoj Plakal, Mark D. Hill, and David A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, November 2000.

[14] 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*, October 1996.

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

[16] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations. In *Proceedings of the 2001 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, June 2001.

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

[18] SPEC. SPEC JBB2000. http://www.spec.org/osg/jbb2000.