# An Architectural Evaluation of Java TPC-W

Harold W. Cain, Ravi Rajwar

*Computer Sciences Department*
*University of Wisconsin*
*Madison, WI 53706*
*{cain,rajwar}@cs.wisc.edu*

Morris Marden, Mikko H. Lipasti

*Dept. of Electrical and Computer Engineering*
*University of Wisconsin*
*Madison, WI 53706*
*{marden,mikko}@ece.wisc.edu*

## Abstract

*The use of the Java programming language for implementing server-side application logic is increasing in popularity, yet there is very little known about the architectural requirements of this emerging commercial workload. We present a detailed characterization of the Transaction Processing Council's TPC-W web benchmark, implemented in Java. The TPC-W benchmark is designed to exercise the web server and transaction processing system of a typical e-commerce web site. We have implemented TPC-W as a collection of Java servlets, and present an architectural study detailing the memory system and branch predictor behavior of the workload. We also evaluate the effectiveness of a coarse-grained multithreaded processor at increasing system throughput using TPC-W and other commercial workloads. We measure system throughput improvements from 8% to 41% for a two context processor, and 12% to 60% for a four context uniprocessor over a single-threaded uniprocessor, despite decreased branch prediction accuracy and cache hit rates.*

## 1. Introduction

In the last few years, the world-wide web has evolved from a global repository for static information into a dynamic environment that provides mechanisms for connecting to and interacting with an ever-increasing number of on-line databases and other sources of dynamic content. In the pursuit of global market share and mindshare, companies ranging from traditional "brick-and-mortar" retailers to online-only startup companies are implementing web sites that allow a high level of dynamic interaction with their inventory, purchasing, and order databases. This transition to online electronic commerce is placing new demands on both the software and hardware infrastructure used to implement these complex systems. On the software side, new implementation techniques like increasing layers of middleware, business object frameworks, and new programming languages such as Java are required to simplify the task of the application programmer and reduce development time. On the hardware side, the overhead of such software techniques and the unique demands created by the confluence of web connectivity, complex middleware and application logic, as well as the high availability expected of traditional on-line transaction processing database systems have the potential to dramatically change the behavior of the program code being executed on the large server computer systems that serve as the hardware platforms for these web sites.

Historically, the Transaction Processing Performance Council (TPC), a consortium of system and database vendors, has specified standard benchmarks (e.g. TPC-A, TPC-B, TPC-C, TPC-D, TPC-H, TPC-R) for evaluating the performance of both transaction processing and decision support database systems [27]. These benchmarks have been very useful for gauging the absolute performance and price/performance of combined software/hardware systems. A significant body of prior work has studied the architectural requirements of such workloads [13,15,17,19,20, 22]. At the same time, the Systems Performance Evaluation Cooperative (SPEC), a similar consortium, has developed standard benchmarks for evaluating both static and dynamic web content serving (SPECweb96 and SPECweb99, respectively) [26]. While useful in their own right, none of these benchmarks reflect the demands placed on systems that must perform all of these functions in concert. As a response to this shortcoming, the TPC has developed a new benchmark for e-commerce called TPC-W that is modeled after an on-line bookstore, and includes many of the elements present in such a website, including complex application logic, a significant web serving component including both static and dynamic web pages, and direct transaction processing and decision support connectivity to an online relational database containing product inventory, customer, and order tracking information. We have implemented most of the requirements specified in the TPC-W specification and have published some of our early findings [4]. An overview of the TPC-W specification is presented in Section 2, and the details of our implementation are

described in Section 3. In Section 4 we present results characterizing this benchmark natively on a six-processor IBM RS/6000 S80 shared-memory multiprocessor system.

Meanwhile, as the proliferation of the world-wide web has dramatically altered the landscape for commercial server software, architectural trends and new techniques have continued to evolve. Some of these trends include: deeper pipelines that are increasingly dependent on accurate branch prediction, the increasing importance of a high-performance memory subsystem, particularly in multiprocessor systems, and the need for architectural techniques to overcome or tolerate high memory and interprocessor communication latencies. One particularly effective technique studied in the literature and subsequently implemented in the IBM AS/400 line of computers is coarse-grained multithreading (CGMT) [1,10,11,12,18]. In CGMT, multiple thread contexts exist within each physical processor, and thread-switch logic is used to swap threads in and out whenever long-latency events such as cache misses stall the execution of a particular thread. In Section 5, we characterize overall system performance as well as memory subsystem and branch predictor behavior in the context of conventional and CGMT processors in uni- and multiprocessor configurations running the TPC-W workload. As has been shown in the past for other commercial workloads, we find that CGMT is an effective technique for increasing throughput of Java server workloads, despite an increase in cache misses and reduced accuracy of a shared branch prediction mechanism.[1]

## 2. TPC-W Benchmark Specification

The TPC-W benchmark is a transactional web benchmark which models an on-line bookstore. The benchmark comprises a set of operations designed to exercise a web server/database system in a manner representative of a typical internet commerce application environment. This environment is characterized by multiple concurrent on-line browser sessions, web serving of static and dynamically generated web pages, and a variety of access and update patterns to a database consisting of many tables with different sizes, attributes and relationships. We begin this section with a general overview of the benchmark, followed by a detailed description of the web serving and database components of the workload.

1. We emphasize that our implementation does not fully conform to the TPC-W specification, and hence none of the results presented in this paper should be interpreted as accurate representations of the performance of the software or hardware systems described in this paper. Our results do not meet the strict reporting and auditing requirements specified by the TPC, are in no sense official or comparable to any other TPC-W results, and should not be considered as such.

### 2.1. TPC-W Overview

The components of TPC-W can be logically divided into three tiers: a set of emulated web browsers, a web server, and a means of persistent storage. The emulated browsers simulate the activities of multiple concurrent web browsing users, each making autonomous requests to a web server for both web pages and images. Depending on the web page requested, it may be necessary for the web server to communicate with the persistent storage mechanism and dynamically generate a response page. The persistent storage mechanism records all of the data necessary for an online bookstore (e.g. inventory, customer records, order records, etc.). Although the web server and storage mechanism are logically separated into two parts, the benchmark specification does not preclude merging them.

The TPC-W specification defines 14 web interactions, each different from one another in terms of the required amount of server-side processing. Some are relatively light-weight, requiring only web serving of static HTML pages and images. Others require a considerable amount of server-side processing, involving one or more connections with a database in addition to dynamically generating HTML pages. A few characteristics related to the amount of processing required by each web interaction are summarized in Table 1. Most of the web interactions require dynamic HTML generation, including all of those which communicate with the database. The amount of work done by the database varies from one interaction to another, and while many of the interactions perform simple selects and updates to the database, others (e.g. Best Seller and Admin Confirm) perform complicated transactions similar to those found in decision support system workloads. Table 1 lists the number of joins required for each interaction's queries, as an approximation of the query complexity. There is also a number of static images associated with each web interaction, ranging in size from 5KB to 1MB. The last column in this table shows the response time requirement for each of the web interactions. At least 90% of each type of web interaction must complete within this maximum response time.

A TPC-W benchmark run begins by starting a certain number of emulated browsers, each of which begins a browsing session at the TPC-W bookstore home page. These emulated browsers continue traversing the bookstore's web pages, following different links and entering user information with varying probabilities. The number of emulated browsers used is variable, and determines the maximum reported throughput results. During an emulated browser session, the browser may fill a shopping cart, perform searches in the inventory database, fill out customer information, perform bookstore administrative duties, buy the contents of the shopping cart, look at best sellers and

| Name | Dynamic HTML? | # Table Joins | # Images | Max Response Time(seconds) |
|---|---|---|---|---|
| Admin Confirm | Yes | 4 | 5 | 20 |
| Admin Request | Yes | 2 | 6 | 3 |
| Best Seller | Yes | 3 | 9 | 5 |
| Buy Confirm | Yes | 1 | 2 | 5 |
| Buy Request | Yes | 1 | 3 | 3 |
| Customer Registration | No | N/A | 4 | 3 |
| Home | Yes | 1 | 9 | 3 |
| New Product | Yes | 2 | 9 | 5 |
| Order Display | Yes | 1 | 2 | 3 |
| Order Inquiry | No | N/A | 3 | 3 |
| Product Detail | Yes | 2 | 6 | 3 |
| Search Request | No | N/A | 9 | 3 |
| Search Result | Yes | 2 | 9 | 10 |
| Shopping Cart | Yes | 1 | 9 | 3 |

**Table 1: TPC-W Web Interaction Characteristics.**

new products lists, and make inquiries about previous orders. There is a random period of time spent sleeping between subsequent individual browser requests, to simulate a user's think time. The emulated browsers continue to access the system under test for several minutes. Once the system reaches a steady state, performance measurements begin.

The primary performance metric tested by the TPC-W benchmark is throughput, measured as the number of web interactions per second (WIPS). The specification defines three different mixes of web interactions, each varying the ratio of inventory browsing related web pages visited to ordering related web pages visited. Depending on the particular mix that is used, a remote browser emulator is more or less likely to visit certain parts of the store-front website. The primary mix used is the shopping mix (throughput is denoted as WIPS), which is intended to reflect an average shopping scenario, in which 80% of the pages a user visits are related to browsing and 20% of the pages are related to ordering. Because actual usage patterns may vary for different web sites, the TPC-W specification defines two other web interaction mixes: a browsing mix (WIPSb) in which very little (5%) ordering occurs, and an ordering mix (WIPSo), in which the ratio of browsing to ordering is even. The exact web interaction frequencies for each mix are shown in Table 2. Results are reported for each of the three web interaction mixes, and the TPC specifies that all hardware and software configuration must be identical for each of the interaction mixes.

| Web Interaction | Browsing Mix (WIPSb) | Shopping Mix (WIPS) | Ordering Mix (WIPSo) |
|---|---|---|---|
| **Browse** | **95%** | **80%** | **50%** |
| Best Sellers | 11.00% | 5.00% | 0.46% |
| Home | 29.00% | 16.00% | 9.12% |
| New Products | 11.00% | 5.00% | 0.46% |
| Product Detail | 21.00% | 17.00% | 12.35% |
| Search Request | 12.00% | 20.00% | 14.54% |
| Search Results | 11.00% | 17.00% | 13.08% |
| **Order** | **5%** | **20%** | **50%** |
| Admin Confirm | 0.09% | 0.09% | 0.11% |
| Admin Request | 0.10% | 0.10% | 0.12% |
| Buy Confirm | 0.69% | 1.20% | 10.18% |
| Buy Request | 0.75% | 2.60% | 12.73% |
| Customer Registration | 0.82% | 3.00% | 12.86% |
| Order Display | 0.25% | 0.66% | 0.22% |
| Order Inquiry | 0.30% | 0.75% | 0.25% |
| Shopping Cart | 2.00% | 11.60% | 13.53% |

**Table 2: Web Interaction Frequencies for Each Mix**

## 2.2. Database Component

The TPC-W specification defines the exact schema used for the database. This schema consists of eight tables: customer, address, order, order line, credit card transaction, item, author, and country. At a minimum, a TPC-W implementation must include a database whose tables contain the exact organization specified by this schema. However, additional fields and tables may be added. Most of the benchmark tables are modified over the course of an execution; only the author and country tables are read-only. The ratio of read-only to read-write queries changes for each of the different mixes. The queries associated with the browsing-related web interactions are all read-only, while the order-related web interactions contain many database updates.

The scaling rules for the database are dependent on two variables: the size of the bookstore inventory and the number of emulated browsers that will be used to drive the system. The number of items in the item table is chosen from one of five predetermined sizes. The number of emulated browsers may be scaled in increments of one. The database scaling rules are shown in Table 3, along with typical row lengths and table size estimates for 1,000 emulated browsers and 10,000 items In addition to storing these tables, the system under test must also store images and image thumbnails associated with each item.

The TPC-W specification defines a set of consistency requirements which the system under test must pass prior

| Table Name | Cardinality (in rows) | Typical Row Length (bytes) | Typical Table Size (bytes) |
|---|---|---|---|
| CUSTOMER | 2880 * (number of EB) | 760 | 2,188,888k |
| ADDRESS | 2*CUSTOMER | 154 | 887,040k |
| ORDERS | .9 * CUSTOMER | 220 | 570,240k |
| ORDER_LINE | 3*ORDERS | 132 | 1,026,432k |
| CC_XACTS | 1*ORDERS | 80 | 207,360k |
| ITEM | 1k, 10k, 100k, 1M, 10M | 80 | 207,360k |
| AUTHOR | .25*ITEM | 630 | 1,575k |
| COUNTRY | 92 | 70 | 6.44k |

**Table 3: Database Scaling Rules and Example Table Sizes**

to publishing official results. Four of these requirements pertain to database transactions and are called the ACID requirements. The ACID requirements specify the necessary conditions of atomicity, consistency, isolation and durability for the system. The atomicity requirement guarantees that for a given web interaction, either all database operations occur or none. The consistency requirement specifies that given an initially consistent database, any TPC-W database transaction transitions the database from one consistent state to another. The isolation requirement specifies that all TPC-W interactions must be isolated from one another, meaning that concurrent transactions must yield the same results as serialized transactions. The durability requirement specifies that all database transactions must be durable, and the system will preserve the effects of a committed database transaction after recovery from any single point of failure.

### 2.3. Web Server Component

The web server software must include logic for tracking user sessions. Each emulated browser session must be tracked with a session identification number, meaning that each request made by an emulated browser must include this session identifier. Web servers and browsers typically use one of two methods to maintain and communicate sessions. The first is through cookies, in which the web server responds to a browser's initial request with a cookie, which the browser includes with all subsequent requests. The second means of maintaining sessions is through URL rewriting, in which the web server encodes a session ID as part of the links within a HTML document. When a browser follows one of these links, the session ID is communicated as part of the requested URL. The disadvantage of using URL rewriting is that every web page must be dynamically generated in order to properly encode the session ID in all of the web page's links.

In addition to serving static and dynamically generated HTML documents, the web server must also serve the images referenced in each of these documents. The storage requirements associated with these images amount to approximately 25 kilobytes per inventory item. The image file set may either be stored in the database or on a standard file system.

In order to provide secure on-line payment authorization, the web server must include support for secure sockets layer (SSL) communications. The benchmark defines a payment gateway emulator which authorizes the payment of the purchasing transaction for the buy confirm web interaction. This payment gateway emulator is a program external to the system under test, with which the web server must communicate prior to completing the buy confirm interaction.

Similar to the database consistency requirements, the web server must also follow strict consistency requirements. These requirements affect the amount of allowable web page caching, specifying that any update made to the system must be reflected in pages returned by the web server at most thirty seconds after the update. Prior to this thirty second limit, either all of the effects of the update must be made visible or none. The search results page is the only web page which is exempt from this thirty second limit. Search results are minimally required to reflect the state of the system after initial population. This relaxation was introduced to the specification to permit the use of commercially available web searching and indexing tools.

## 3. A Java Implementation of TPC-W

The TPC-W specification dictates a set of requirements which an implementation must meet, but it allows substantial freedom for making various implementation decisions. We have implemented all of the TPC-W application logic in Java using the Java Servlet API. Very little has been published concerning the behavior and architectural characteristics of this new workload. In this section, we will discuss some of the characteristics of our implementation and their implications on performance. We also describe the areas in which our implementation strays from the specification.

Our implementation's most notable feature is the use of Java for all bookstore application logic. We have implemented each of the 14 web interactions as a Java servlet using the Java Servlet API, an alternative to the traditional common gateway interface (CGI) used by web servers for dynamic HTML generation and server-side application processing. The servlet interface provides a standard, portable programming environment for writing server-side applications. The servlet API provides Java libraries for receiving and responding to HTTP requests and maintaining state for each user session. The portable nature of Java has enabled the use of these servlets on multiple platforms; in a previous study, we were able to do a preliminary char-

acterization of this workload on two architectures with no changes to the benchmark application [4]. For this study we use the Zeus web server with the Apache Jserv Java servlet engine.

For the database component of this workload, we use the IBM DB2 relational database system. Database query and update requests are made by our servlets using the Java Database Connectivity (JDBC) API. As described in Section 2, the TPC-W specification organizes application data into eight tables. In addition to the schema required by the specification, our implementation stores the user's shopping cart in the database because the shopping cart must be durable across any single point of failure for up to two hours. We added two tables to the database to fulfill this durability requirement. One advantage to keeping this state in the database is that it removes all state associated with a specific user session from the web server and Java servlet engine. The elimination of this state allows us to use multiple independent web servers and servlet engines for handling requests. If necessary, different requests for any user session may be routed to one of many servers, each communicating with the DB2 back-end.

Although the Servlet API provides library methods for tracking sessions using both cookies and URL rewriting, our implementation relies solely on URL rewriting. By restricting ourselves to URL rewriting for session tracking, we avoid the complexity of managing cookies in our remote browser emulator. Unfortunately, this means that all HTML documents must be dynamically generated in order to encode a session identifier in each link. Eleven of the fourteen web interactions require dynamically generated HTML to display database query results, so there are only three web interactions in which we pay a penalty for doing URL rewriting. Consequently, we believe that our choice of URL rewriting does not substantially affect the performance of our implementation.

Our implementation of the TPC-W benchmark strays from the specification in the following ways: we do not use the secure sockets layer connection required for the buy confirm interaction, or do the credit card authorization associated with this web interaction. According to the specification, the system under test must communicate credit card information to an external application for payment authorization during the buy request web interaction, and must also use the secure sockets layer for communicating credit card information. By neglecting to implement this feature, the web server does less work for the buy confirm interaction than required by the specification. For the browsing and shopping interaction mixes, this interaction accounts for less than two percent of all web interactions, and ten percent for the ordering mix. Because of its relative infrequency, we believe that the loss of this functionality does not seriously affect the behavior of the workload.

# 4. Native Execution Results

In this section, we present the results of experiments performed while executing the workload natively on a 6-processor IBM RS/6000 S80 SMP system running the AIX 4.3.3 operating system. We find that most memory system stalls are due to L2 cache load misses, and that a large fraction of L2 misses are serviced by cache-to-cache transfers. We also show the importance of using the exclusive state in cache coherence protocols when executing TPC-W as well as other commercial workloads.

## 4.1. Workloads

To better understand how the behavior of TPC-W compares to other current server benchmarks, we also present results for SPECweb99 and SPECjbb2000.

SPECweb99 is a web serving benchmark which tests many of the requirements of modern web servers, such as static and dynamic HTTP requests, keep alive and persistent connections, and dynamic advertisement rotation using cookies. Our SPECweb99 results were collected using the same Zeus web server used for the TPC-W experiments.

The SPEC Java Business Benchmark (SPECjbb) 2000 is designed to test server-side Java performance. The benchmark is written entirely in Java, and runs on a single instance of a Java virtual machine. It emulates a three-tier system similar to TPC-C in which a wholesale company containing multiple warehouses services many concurrent users. System throughput scales with the number of warehouses used. We use six warehouses for these experiments, and the same JVM for running both SPECjbb2000 and the TPC-W Java servlet engine.

## 4.2. Methodology

The IBM RS/6000 S80 is a shared-memory symmetric multiprocessor based on the PowerPC RS64-III (Pulsar) microprocessor [5, 25]. Pulsar is a four-issue in-order superscalar processor with a five stage RISC pipeline. Pulsar tolerates memory latency with coarse-grained multithreading by switching to an alternate thread whenever the currently running thread encounters a cache miss. If there is no other ready thread, the pipeline is stalled until the miss resolves. Unfortunately the multithreading feature on Pulsar is disabled in AIX 4.3.3. Despite the lack of multithreading in the S80, until recently (July 2000) this system was the non-clustered TPC-C performance leader [27]. We present simulation results evaluating coarse-grained multithreading in Section 5.

Considerable effort has been spent tuning this workload, and our system setup represents the state-of-the-art in hardware and software for running commercial server work-

| Hardware | |
|---|---|
| Processors | 6-way 450 MHZ RS-64 III (Pulsar) |
| Memory | 8 GB |
| L2 Cache | 8 MB unified cache per processor |
| L1 Cache | 128 KB I-Cache, 128 KB D-Cache |
| **Software** | |
| IBM JDK 1.1.8 with JITC, AIX 4.3.3 | Apache Jserv 1.0 |
| IBM DB2 6.1 | Zeus Web Server 3.3.7 |
| Database Size: 204 MB | Image Set Size: 250 MB |

**Table 4: System Parameters**

loads [14]. A few pertinent system parameters are described in Table 4.

The smallest allowed item table size is 1,000 items; we use a larger configuration with the scaling rules for 10,000 items and 50 emulated browsers.

The RS-64 III processor includes a rich set of eight 64-bit performance counters. In combination, these counters can record over 275 unique events ranging from memory system statistics such as cache misses, coherence protocol transitions, and bus utilization, to microarchitectural issues such as coarse grained multithreading events and branch penalties. Using these counters, we have collected substantially more data than have room to present here; a more complete set of data can be found in an extended version of the paper [6].

The software used for controlling these counters supports counting on the granularity of a process tree, where a single set of virtual counters is shared by a parent process and all of the its child processes. We use this process tree counting mode for all natively-collected results. Counting is enabled for both user and system-level instructions.

While performing these experiments, the remote browser emulators are run on workstations external to the system under test, with the browser think time set to zero seconds in order to place a heavy load on the server while using fewer client CPU resources. A similar strategy was used in Baylor et al. [3]. For each workload, the measurements are performed during several runs consisting of a 15 minute warm-up phase followed by a 60 minute measurement phase.

## 4.3. CPU Utilization by Component

Figure 1 shows the measured CPU utilization while running TPC-W for each of the three web interaction mixes, breaking down the CPU utilization into parts for each of the three workload components. We see that the Java servlet engine component dominates the CPU usage, accounting for 80% of the CPU utilization in each of the web interaction mixes. The database accounts for the majority of the remaining CPU time, with web server time accounting for less than 3% of the CPU in all mixes. We see some idle time during the update-intensive ordering mix because of synchronization within the servlet implementation which prevents conflicting updates to the database.

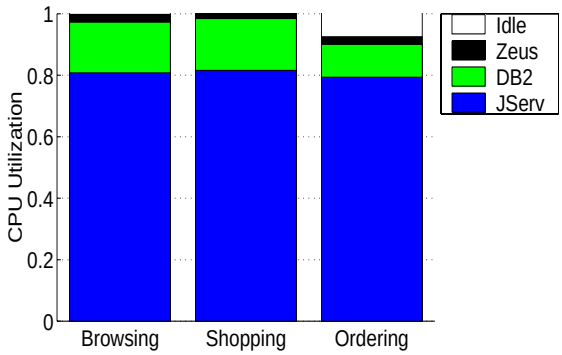



**Figure 1. CPU utilization divided among web server, database, and servlet engine**

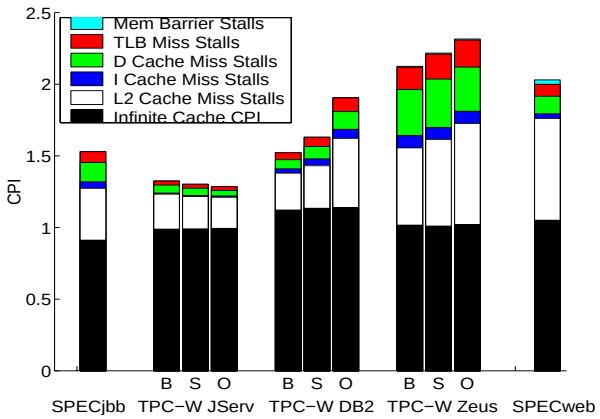



**Figure 2. CPI Breakdown - the TPC-W runs are broken into groups of three bars, where each group corresponds to a different component of the workload, and B, S, O bars within a group correspond to browsing, shopping and ordering mix runs.**

## 4.4. Memory System Characterization

Figure 2 shows the measured CPI for each of the workloads, with memory system stall time broken into its components. In the absence of cache misses, the processor runs at a CPI near one in all cases. The memory system contribution to the CPI varies considerably among workloads, with TPC-W Zeus and SPECweb99 most affected by memory system stalls, and the Jserv and SPECjbb Java applications least affected. Zeus is the only TPC-W component which touches the static image file set, and DB2 has a large memory footprint, which account for increased memory system related stalls for the two applications. The TPC-W servlet engine suffers relatively few memory system stall cycles; its primary function is formatting and forwarding requests and responses between the database and emulated

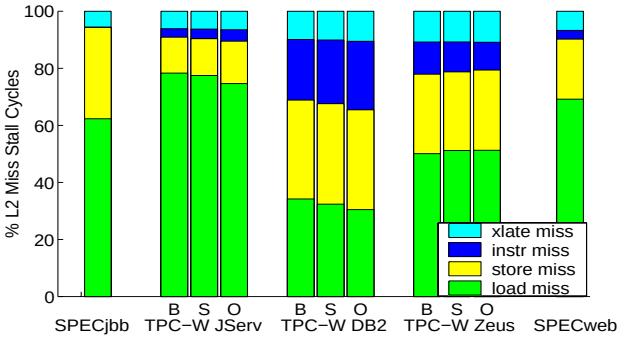



**Figure 3. Breakdown of L2 Cache Miss Stall Cycles**

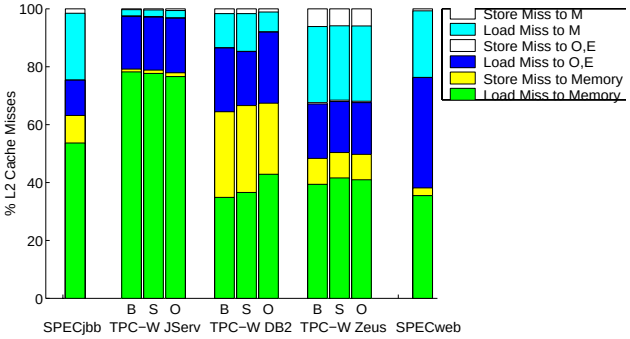



**Figure 4. Breakdown of Cache Line Residency forL2 Cache Misses**

browsers, so it does not touch the volumes of data managed by the database and web server, consequently resulting in fewer memory system stalls. In all cases, level two cache misses account for the majority of memory related stall cycles.

Figure 3 further breaks down the L2 stalls category into L2 miss stalls caused by instructions, loads, stores, and translations. L2 translation stalls occur when the PowerPC hardware page fault handler loads page table entry groups into the cache while traversing the page table. As one would expect, load misses dominate the number of L2 cache miss stall cycles, however DB2 also shows a considerable number of L2 store miss stalls. Although the database and web server applications are affected by L2 instruction misses, the servlet engine and SPECjbb2000 suffer almost no instruction related L2 stalls.

## 4.5. Cache-to-Cache Transfers

Previous studies of commercial workloads have demonstrated the importance of optimizing cache-to-cache transfers due to the high percentage of cache misses to lines resident in the caches of other processors [2,15,17,22]. We present a breakdown of cache misses based on the location from where they are serviced in the S80 memory system. The S80 server implements a MOESI coherence protocol where cache misses to blocks in another processor's cache are serviced by that processor when a cache block is in the Owned (O), Exclusive (E), or Modified (M) state.The results of these measurements are shown in Figure 4. Level 2 cache misses are broken down into those misses which are serviced from memory, and those misses which are serviced by other caches containing a copy of the line in the M, O or E state. Our data confirms previous work for the web serving and database portions of the workload, and also shows that a substantial fraction of references are serviced by another cache in the dirty state in the Java server workloads as well. In general, we see that at least 20% of all L2 cache misses are serviced by cache-to-cache transfers, with as many as 52% to 62% in SPECweb99 and the web serving component of the TPC-W workload. Of these

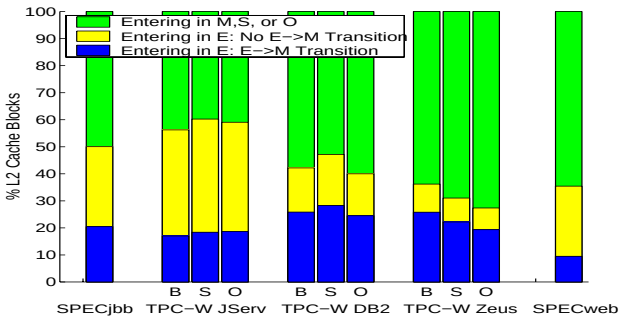



**Figure 5. Use of the Exclusive State - bars show the percentage of all cache lines entering the L2 cache in the E state, and the percentages of those which make transitions to the M state.**

cache-cache-transfers, between 40% and 60% are dirty misses in Zeus, DB2, and SPECjbb2000, while almost none of the cache-to-cache transfers are dirty misses in the Java servlet engine.

## 4.6. Impact of the Exclusive State

Recent work characterizing commercial workloads has argued that the exclusive state in multiprocessor cache coherence protocols is not useful since stores rarely (less than 2%) find cache lines in the L2 cache in the exclusive state (E state) [7, 15]. We believe that the percentage of stores hitting the L2 cache in the E state is not a good indicator of the E state's usefulness. Due to spatial locality, many stores may reference a cache line, yet only the first store to the line will find it in the E state. This single transition count is a processor-centric view of the E state; because most processor references are filtered by the L2 cache, this count does not reflect the demands placed on the system-wide interconnect.

A better indicator of E state usefulness is the frequency with which cache lines enter the cache in the E state and transition to the M state. As shown in Figure 5, we see that between 27% and 60% of all cache lines enter the L2 cache in the E state. Furthermore, between 26% and 71% of these cache lines eventually make the transition to the M state. Because each transition from the E state to the M state

saves a single bus upgrade transaction, we find that an extra bus transaction would be required for as many as 28% of all L2 cache misses in a simple MSI protocol. For this reason, we believe adding the E state to multiprocessor cache coherence protocols is justified.

## 5. Simulation Results

Although much can be learned about the behavior of a system using performance monitoring hardware, we are restricted to studying existing systems. If one would like to study the performance implications of new architectural techniques without building hardware, simulation is necessary. In this section, we present a study of the effects of coarse-grained multithreading on system performance, results which could not have been collected using performance counters. The use of multithreading affects many aspects of processor design, because the access of shared resources by different threads may have an impact on each thread. In Section 5.2, we contrast the behavior of several different branch predictor designs in the presence of multithreading. In Section 5.3, we examine the effects of multithreading on cache behavior and sharing. In Section 5.4, we measure the performance impact of coarse-grained multithreading on the TPC-W workload.

For the experiments presented in this section, we include results for the SPECint_rate95 benchmark in addition to TPC-W, SPECjbb2000, and SPECweb99. SPECint_rate95 consists of concurrently executing all of the SPEC95 integer benchmarks as a measure of system throughput.

### 5.1. Methodology

We use an augmented version of the SimOS-PPC [16] full system simulator, which is a PowerPC port of the simulator originally developed at Stanford University [23]. SimOS-PPC runs a slightly modified version of AIX 4.3.1. Given the level of interaction between system code and user code and the amount of inter-process communication in these workloads, the ability to simulate user and system level code is essential. SimOS-PPC faithfully simulates all system devices in detail.

SimOS-PPC uses the same setup as the native execution runs in terms of disks, database setup, and applications. Thus, all the software parameters for the workload carry over from the native runs. The slow speed of detailed processor and memory system simulation constrains the length of time the workload is actually studied. We use the high speed simulation mode of SimOS-PPC to warm up the workload. With the exception of SPECint_rate95, we use a three second snapshot of steady-state behavior (after a warm-up period) for timing statistics. SPECint_rate95 is run to completion using reduced input sets.

Our processor model approximates the behavior of the RS64-III (Pulsar) processors used in IBM RS/6000 S80 systems. We collect results for one thread per processor (no CGMT), two threads per processor, and four threads per processor configurations; Pulsar implements two threads per processor. As shown in Section 4.4, Pulsar runs at approximately 1.0 cycles per instruction in the absence of cache misses; hence, we do not model the details of the pipeline but instead charge one cycle for every instruction that is executed and use an accurate memory subsystem timing model that accounts for latencies and contention in the memory hierarchy.

Our simulator models a simplified version of the Pulsar thread switch state machine. There are four possible states for a thread: running, ready, stalled, and swapped. Threads transition between states whenever a cache miss is initiated or completed, and when the thread switch logic decides to switch to an alternate thread. The conditions for switching to an alternate thread are the following:

- A cache miss has occurred in the primary thread, and there is another thread in the ready state.
- The primary thread has entered the idle loop or is spinning on a lock, and there is a non-idle, non-spinning thread in the ready state.
- An alternate thread has a pending interrupt or exception.
- An alternate ready, non-idle, thread has not retired an instruction in the last 1000 cycles.

Forward progress is guaranteed by preventing a preemptive thread switch from occurring if the running thread has been active for less than 100 cycles. We assume a 3-cycle thread switch penalty for draining the pipeline and resuming instruction fetch from the new active thread; this matches the Pulsar switch penalty.

We model an aggressive 4-way snooping-bus shared memory multiprocessor. The snoop design is modeled after the Sun Gigaplane [24], which uses a split-transaction, pipelined address bus with support for 120 outstanding transactions and out-of-order responses. The bus implements an invalidation-based 3-state (Owned, Shared, Invalid) snooping cache coherence protocol with no transient states and the level two caches implement the MOESI protocol. The data network is modeled after the Sun Gigaplane-XB [8] and is a 64-bit wide point-to-point data crossbar. The latencies in the memory system are modeled in detail, as is contention at all levels of the memory hierarchy. Due to the differences between these parameters and the S80 memory system, and the difference of processor counts, the results presented in this section are not comparable with those presented in Section 4. All other memory system parameters match those in Table 4.

For those workloads which use a driver program, we bind the driver program to an extra processor on which we

do not collect statistics. For example, in the 4-way SMP TPC-W runs, the browser emulator is run on a fifth processor for which no statistics are collected.

## 5.2. Effect of Multithreading on Branch Prediction

We simulate three different GShare branch predictor [21] configurations with branch history tables (BHT) ranging from 1K to 256K entries. Figure 6 shows the branch misprediction rates of 1,2, and 4-way multithreaded uniprocessor systems for the six workloads. Note that our simulator ignores branches in idle loops and unconditional branches.

We find that when the threads on a processor share a standard branch predictor, as shown in the shared BHR and BHT case in Figure 6, the histories of the branches destructively interfere with one another, negatively affecting the predictor's accuracy. To improve the predictor's performance in multithreaded processors, we simulate two new types of branch predictors. We first examine separate branch predictors for each thread, where the size of each of the predictors is scaled down by the number of threads. Thus, in the case of a dual threaded processor, the two branch predictors are one-half the size of the single threaded processor's branch predictor. Likewise, the processor with four threads has four branch predictors one-fourth the size of the single threaded processor's branch predictor. Our results show that this is an effective method for reducing the number of mispredictions caused by interference between threads. However, since each individual branch predictor is smaller in size than the single threaded processor's branch predictor, the individual branch predictors still perform worse than the large uniprocessor branch predictor.

A disadvantage of using separate branch predictors for each thread is that this strategy cannot adapt to workloads which require branch predictors of varying sizes. For example, if one thread has a larger BHT than it needs, it cannot share entries of its BHT with a second thread whose working set does not fit in the BHT.

To address this problem, we also simulate a branch predictor which has a separate branch history register (BHR) for each thread but shares the BHT between all threads. Our results show that in all of the workloads except SPECweb99, this branch predictor performs better than separate smaller branch predictors for each thread when using small branch predictors. However, this branch predictor still performs significantly worse than the large private branch predictor used in the uniprocessor run, indicating a substantial amount of conflict among threads

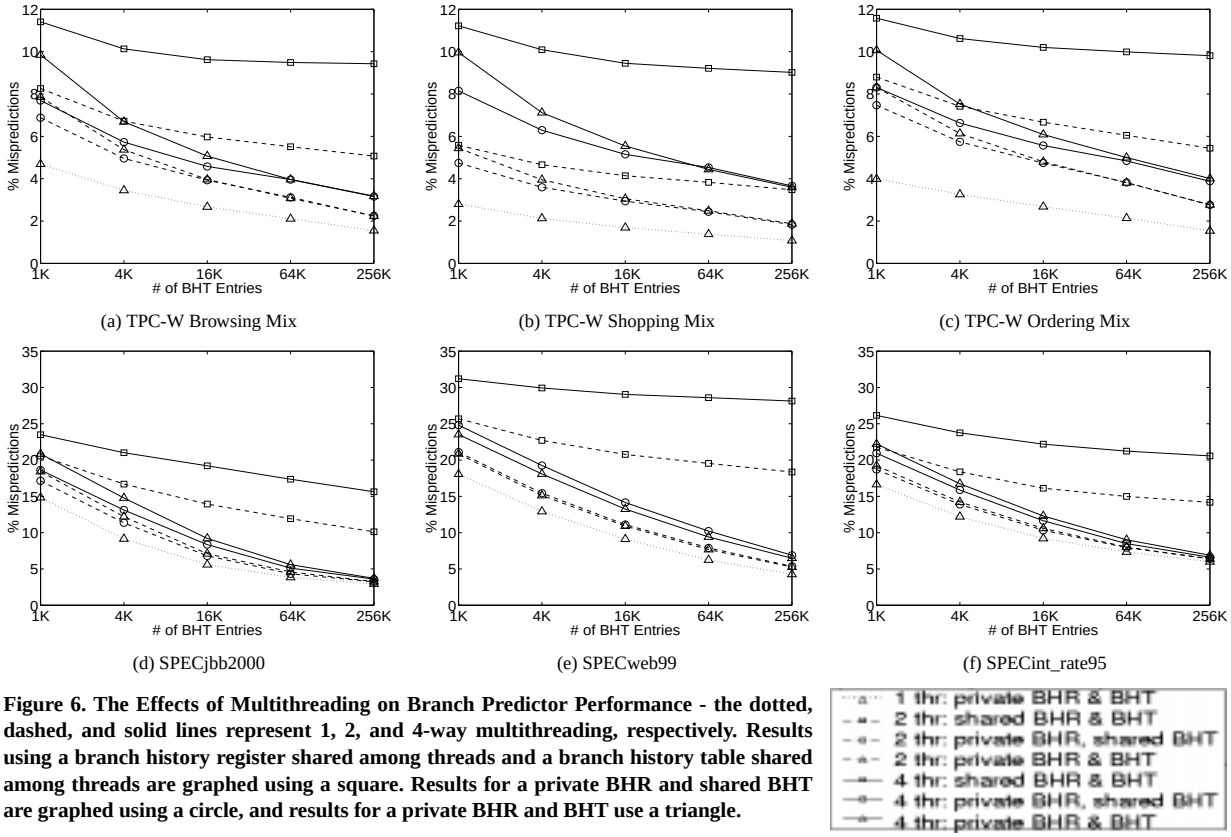
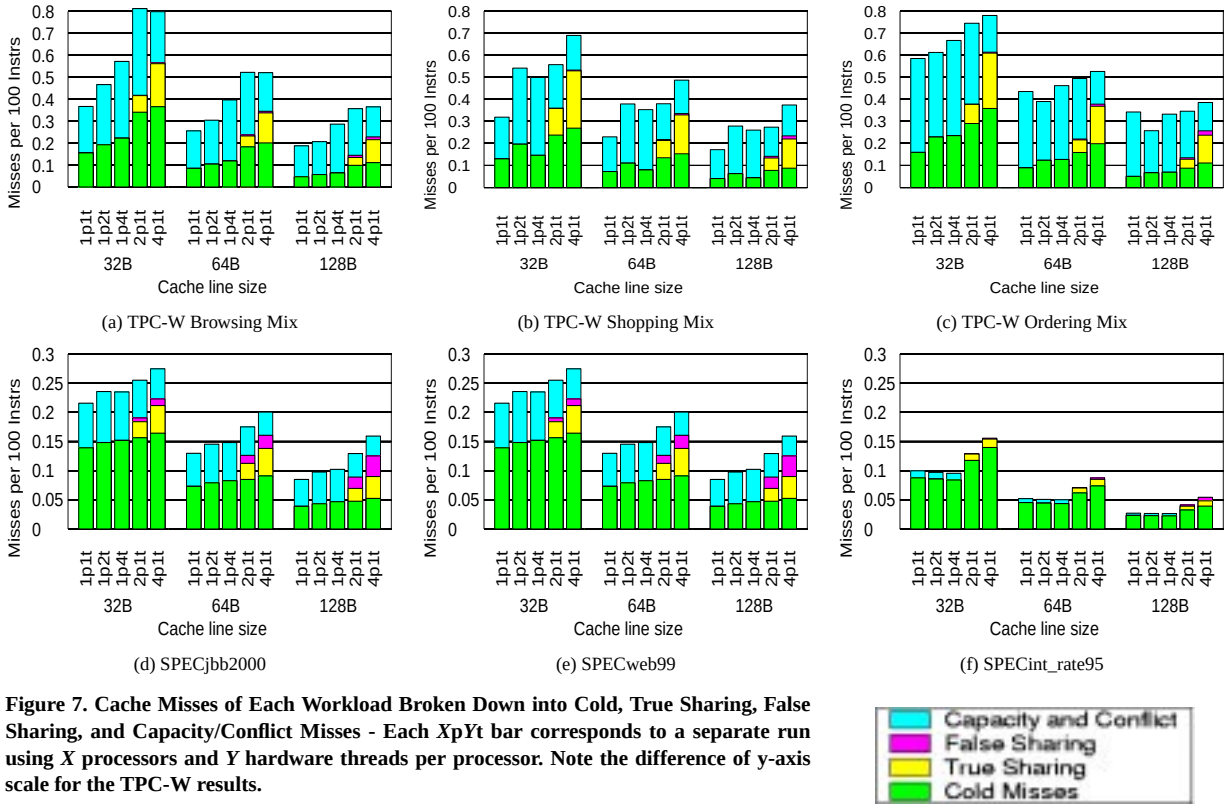



**Figure 6. The Effects of Multithreading on Branch Predictor Performance - the dotted, dashed, and solid lines represent 1, 2, and 4-way multithreading, respectively. Results using a branch history register shared among threads and a branch history table shared among threads are graphed using a square. Results for a private BHR and shared BHT are graphed using a circle, and results for a private BHR and BHT use a triangle.**




(a) TPC-W Browsing Mix (b) TPC-W Shopping Mix (c) TPC-W Ordering Mix

(d) SPECjbb2000 (e) SPECweb99 (f) SPECint_rate95

**Figure 7. Cache Misses of Each Workload Broken Down into Cold, True Sharing, False Sharing, and Capacity/Conflict Misses - Each *X*p*Y*t bar corresponds to a separate run using *X* processors and *Y* hardware threads per processor. Note the difference of y-axis scale for the TPC-W results.**

in the BHT. As the size of the branch predictor increases, the performance of a private BHT/private BHR predictor-catches up with the shared BHT/private BHR predictor, because each private BHT becomes large enough to minimize conflicting updates to the BHT by different branches.

### 5.3. Effects of Multithreading on Cache Behavior

To measure cache performance, we simulate single level unified four way set associative 8 MB caches with three different line sizes. At the beginning of the simulations, the caches are empty. Note that memory references from instructions in idle loops are not included, since they will usually hit in the cache and thus would artificially lower the cache miss rates. We compute true and false sharing in the caches using Dubois' definition of sharing [9]. The results of the cache simulations are shown in Figure 7. In the multithreaded uniprocessor systems, the number of capacity and conflict misses increase with the number of threads per processor for all of the workloads except for SPECint_rate95. This is not surprising, since all of the threads share the same cache, leaving each thread with effectively a smaller cache than in the case of a single threaded processor. In SPECint_rate95, the 8 MB cache easily contains most of the memory footprint, even when shared among four different threads.

In most cases the number of cold misses grows with the number of threads, which is an artifact of our simulator starting with unwarmed caches. As the number of threads increases per processor, so does the relative percentage of our three second snapshot which is spent warming up the caches. If there is any prefetching occurring among threads, its gains are offset by the increased amount of time spent warming the cache.

Unlike the uniprocessor systems, we see the number of capacity and conflict misses decrease with the number of processors in multiprocessor configurations. Since each processor has its own cache, the aggregate cache size for the entire system is much larger. In all of the workloads, there is a significant amount of true sharing, even for the multiprocessor SPECint_rate95, due to effects of the operating system.

TPC-W exhibits relatively few false sharing misses, despite its much larger number of misses per instruction. In our implementation of TPC-W, there is no shared data among servlets running in the Java servlet engine; all sharing occurs in the database. DB2 is a mature application which has been optimized to reduce false sharing. We believe SPECjbb2000 and Zeus are less mature applications which not been optimized to reduce false sharing, hence the relatively larger number of false sharing misses.

### 5.4. Multithreading Performance

We collect overall cycles-per-instruction (CPI) performance data from our detailed memory subsystem simulator, which is presented in Table 5. These measurements do not include instructions executed while in the idle loop, so they are more precisely a measurement of cycles-per-useful instruction.

| Benchmark | 1p1t | 1p2t | Speedup | 1p4t | Speedup |
|---|---|---|---|---|---|
| SPECjbb2000 | 1.33 | 1.23 | 8% | 1.14 | 17% |
| TPC-W Browsing | 1.58 | 1.42 | 11% | 1.28 | 23% |
| TPC-W Shopping | 2.07 | 1.46 | 41% | 1.33 | 56% |
| TPC-W Ordering | 2.04 | 1.56 | 31% | 1.28 | 60% |
| SPECweb99 | 1.29 | 1.14 | 13% | 1.15 | 12% |
| SPECint_rate95 | 1.10 | 1.05 | 5% | 1.05 | 5% |

**Table 5: Modeled Cycles Per Instruction (CPI) - this table shows the modeled CPI for each benchmark in the 1,2, and 4-way CGMT uniprocessor, and speedups relative to the 1p1t case.**

We find that the use of coarse grained multithreading offers significant speedups for all of the commercial applications, with SPECint_rate95 showing only small speedups. The five commercial workloads show considerable reductions in CPI in the 2-way multithreaded case, as much as 41% for the TPC-W shopping mix, and more reductions in the 4-way multithreaded case. However, the law of diminishing returns is in effect, as those applications which do not incur many memory system stalls gain little from multithreading. The speedups found in those applications which do suffer large memory system penalties, namely all of the commercial workloads, are compelling evidence of the efficacy of coarse-grained multithreading.

## 6. Conclusions

We show that level two cache misses caused by data references are the primary contributor to memory system stall cycles in Java commercial workloads, like other commercial workoads. We find between 20% and 50% of these misses are serviced by cache-to-cache transfers in a six-processor SMP system using a MOESI coherence protocol. We show that using the exclusive state eliminates an extra bus upgrade transaction which would otherwise be necessary for 8% to 28% of all level two cache misses.

We also present the first execution driven study of a coarse-grained multithreaded processor using commercial workloads. We believe that coarse-grained multithreading is a viable means of increasing system throughput given limited processor resources. We saw between 23% and 60% throughput improvements for the four context uniprocessor over a single-threaded uniprocessor when running TPC-W. We also show that sharing of branch prediction resources among contexts is detrimental to the performance of the branch predictor due to negative interference among threads. The addition of multiple threads to the processor also increases conflict misses and contention in the memory system. Despite these two penalties, the ability to tolerate memory system latency by coarse-grained multithreading makes it an attractive choice for commercial workloads.

## 7. Acknowledgments

This work was supported in part by an IBM University Partnership Award and NSF Grants CCR-0073440 and CCR-0083126. We would like to thank IBM for donating the equipment used in this work, and Steve Kunkel for help with the Pulsar performance counters. We would also like to acknowledge the students of UW-Madison's ECE 902 course during the Fall of 1999 who contributed to our TPC-W implementation, and Milo Martin for pointing out the E state methodology problems in prior work [7,15].

## References

[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B. Lim, and D. Yeung. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[2] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[3] S. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. Munroe. Java Server Benchmarks. *IBM Systems Journal*, 39(1), 2000.

[4] T. Bezenek, H. Cain, R. Dickson, T. Heil, M. Martin, C. McCurdy, R. Rajwar, E. Weglarz, C. Zilles, and M. Lipasti. Characterizing a Java Implementation of TPC-W. Third Workshop On Computer Architecture Evaluation Using Commercial Workloads, January 2000.

[5] J. Borkenhagen and S. Storino. 5th Generation 64-bit PowerPC-Compatible Commercial Processor Design. IBM Whitepaper, http://www.rs6000.ibm.com, 1999.

[6] H. Cain, R. Rajwar, M. Marden, and M. Lipasti. An architectural characterization of Java server workloads on multithreaded processors. Technical Report 1421, University of Wisconsin Department of Computer Sciences, 2000.

[7] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed characterization of a quad Pentium Pro server running TPC-D. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, February 1997.

[8] A. Charlesworth, A. Phelps, R. Williams, and G. Gilbert. Gigaplane-XB: Extending the ultra enterprise family. In *Proceedings of the Symposium on High Performance Interconnects V*, August 1997.

[9] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Use-

less Misses in Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[10] R. Eickemeyer, R. Johnson, S. Kunkel, and J. Rose. An Analysis of Multithreading in PowerPC Processors. Technical report, IBM, 1994.

[11] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, and B. H. Lim. Evaluation of Multithreaded Processors and Thread-Switch Policies. *Lecture Notes in Computer Science*, 1336, 1997.

[12] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of Multithreaded Uniprocessors for Commercial Application Environments. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[13] M. T. Franklin, W. P. Alexander, R. Jauhari, A. M. G. Maynard, and B. R. Olszewski. Commercial workload performance in the IBM POWER2 RISC System/6000 processor. *IBM Journal of Research and Development*, 38(5):555–561, September 1994.

[14] IBM Corporation. The RS/6000 Enterprise Server Model S80 Technology and Architecture. IBM Whitepaper available from http://www.rs6000.ibm.com.

[15] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[16] T. Keller, A. Maynard, R. Simpson, and P. Bohrer. SIMOS-PPC Full System Simulator. http://www.cs.utexas.edu/users/cart/simOS.

[17] S. Kunkel, B. Armstrong, and P. Vitale. System Optimization for OLTP Workloads. *IEEE Micro*, May/June 1999.

[18] K. Kurihara, D. Chaiken, and A. Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

[19] J. Lo, L.A. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[20] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *ACM SIG-PLAN Notices*, 29(11):145–156, November 1994.

[21] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Equipment Corp, June 1993.

[22] P. Ranganathan, K. Gharachorloo, S. Adve, and L. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[23] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: the SimOS approach. *IEEE Parallel and Distributed Technology*, 1995.

[24] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, and E. Hagersten. Gigaplane: A high performance bus for large SMPs. In *Proceedings of the Symposium on High Performance Interconnects IV*, August 1996.

[25] S. Storino, A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A Commercial Multi-threaded RISC Processor. In *International Solid-State Circuits Conference*, 1998.

[26] Systems Performance Evaluation Cooperative. SPEC Benchmarks. http://www.spec.org.

[27] Transaction Processing Performance Council. TPC Benchmarks. http://www.tpc.org.