A performance methodology for commercial servers

by S. R. Kunkel
R. J. Eickemeyer
M. H. Lipasti
T. J. Mullins
B. O'Krafka
H. Rosenberg
S. P. VanderWiel
P. L. Vitale
L. D. Whitley

This paper discusses a methodology for analyzing and optimizing the performance of commercial servers. Commercial server workloads are shown to have unique characteristics which expand the elements that must be optimized to achieve good performance and require a unique performance methodology. The steps in the process of server performance optimization are described and include the following:

1. **Selection of representative commercial workloads and identification of key characteristics to be evaluated.**
2. **Collection of performance data. Various instrumentation techniques are discussed in light of the requirements placed by commercial server workloads on the instrumentation.**
3. **Creation of input data for performance models on the basis of measured workload information. This step in the methodology must overcome the operating environment differences between the instance of the measured system under test and the target system design to be modeled.**
4. **Creation of performance models. Two general types are described: high-level models and detailed cycle-accurate simulators. These types are applied to model the processor, memory, and I/O system.**
5. **System performance optimization. The tuning of the operating system and application software is described.**

Optimization of performance among commercial applications is not simply an exercise in using traces to maximize the processor MIPS. Equally significant are items such as the use of probabilities to reflect future workload characteristics, software tuning, cache miss rate optimization, memory management, and I/O performance. The paper presents techniques for evaluating the performance of each of these key contributors so as to optimize the overall performance and cost/performance of commercial servers.

1. Introduction

The performance of commercial servers is a function of many variables and requires careful design optimization as a key contributor to product success in the marketplace. Complexity in satisfying performance demands across the spectrum of commercial applications arises from two dimensions of the system design space:

1. Commercial workloads exhibit significant variation in their usage of computer resources. Some are more compute-intensive, focusing performance demand on processor power. Others stress the I/O subsystem, interacting with disk, LAN, and other device facilities

©**Copyright** 2000 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/00/$5.00 © 2000 IBM

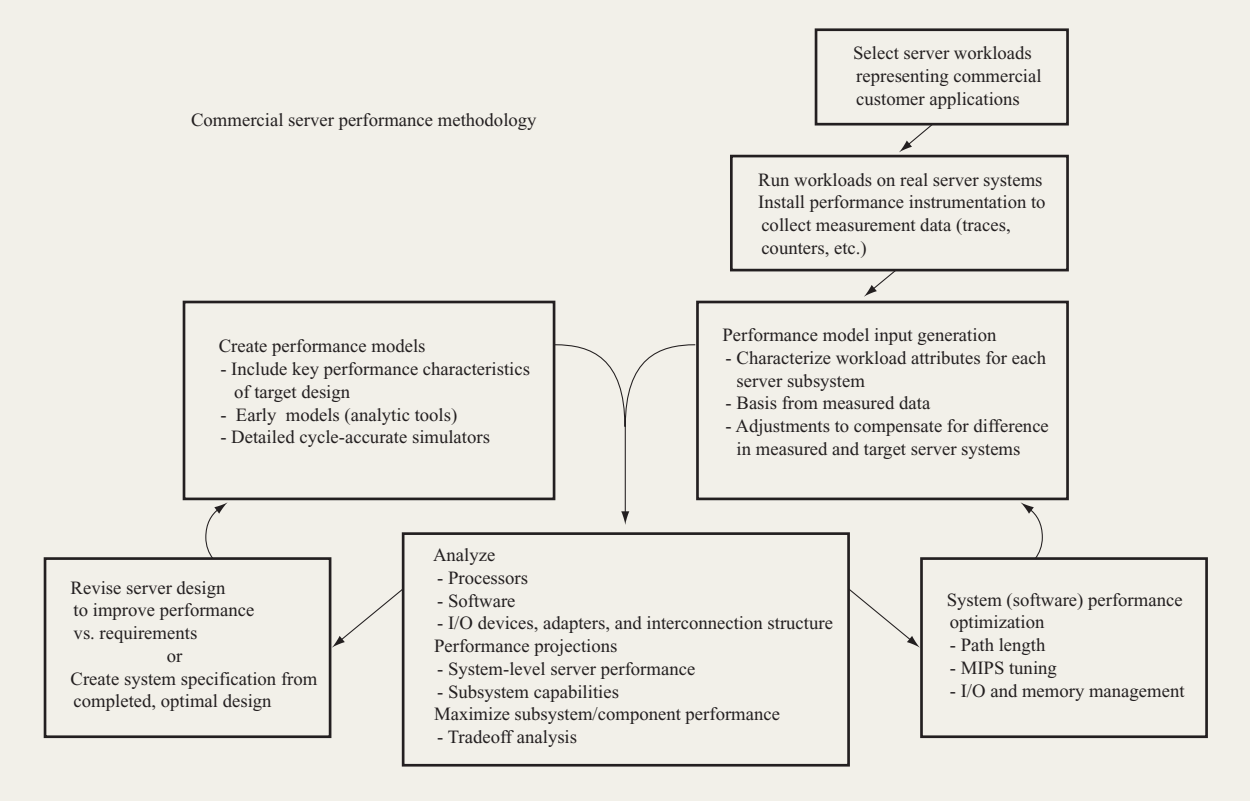

Figure 1

Analysis and tuning of commercial server design.

over system I/O interconnects. Analysis of the performance for the range of commercial workloads must take into account the demands placed on these server subsystems and must evaluate potential bottlenecks in each of them.

2. Interaction of subsystems within a given workload creates performance effects that can be subtle to analyze and difficult to optimize in the system design. Key areas of a commercial server computer that are interrelated include the processor, software (operating system and applications), I/O bus/interconnect, I/O device subsystem, and memory management and usage.

Tradeoffs in the design and optimization of commercial servers must consider the impact on overall system performance when aspects of each of these subsystems are varied and affect other subsystems.

In addition, the scale of commercial servers grows ever larger. Typical configurations of the largest models can involve many tens of thousands of users, tens of gigabytes of main memory, nearly a thousand disk actuators, and tens of terabytes of disk capacity. The extreme scale of these large commercial servers adds even more complexity to the pursuit of optimized performance across the spectrum of workloads.

All of these factors require an extremely challenging effort in understanding, analyzing, and tuning designs to perform well across the breadth of the commercial workload spectrum. The variety of factors that affect commercial server performance require a broad evaluation methodology. Whereas traditional analysis methodologies focus on maximizing processor MIPS, commercial servers must be optimized in many areas to maximize overall system performance for customer applications.

This paper presents the key concepts relating to commercial server performance and describes techniques that have been successfully used in design optimization for IBM servers. A sequence of activities is required to accomplish analysis and tuning of commercial server designs. **Figure 1** illustrates the steps involved. The following sections of this paper expand on the topics referenced in the figure:

Table 1 Attributes of important server workloads.

Benchmark attribute	*TPC-C*	*TPC-H*	*NotesBench*	*SAP 2T*	*SAP 3T DB*	*Server Java*
Intra-thread parallelism	low	high	medium	medium	low	low
Instruction working set	large	medium	medium	medium	large	large
Data working set	large	large	medium	medium	large	large
Data sharing	pervasive	medium	low	medium	pervasive	medium
Contention	high	low	low	low	medium	low
I/O request rate	high	low	low	low	medium	medium
I/O data rate	low	high	very low	very low	low	low

- Section 2 covers commercial workloads and their performance traits.
- Section 3 describes the various approaches to performance instrumentation.
- Section 4 explains the kinds of modification required to adjust raw data acquired from performance instrumentation techniques for use as model input data.
- Section 5 considers high-level modeling techniques and focuses on mean value analysis.
- Section 6 presents details on modeling with cycle-accurate simulators.
- Section 7 relates to an important aspect of commercial server performance—I/O device and adapter effects.
- Section 8 continues the presentation of I/O topics and covers I/O interconnects and their performance attributes.
- Section 9 presents an example of commercial server analysis performed for an IBM server processor.
- Section 10 concludes the paper by addressing system performance optimization through software techniques.
- Section 11 provides a summary of the above topics and a conclusion to the paper.

The remainder of this paper describes the elements of the process flow in more detail. This methodology has proven effective in the design of IBM servers such as the RS/6000* Model S80 (now pSeries* 6000) and the AS/400* Model 840 (now iSeries* 400), with CPU chips including the PowerPC* RS64, RS64-II, RS64-III, RS64-IV, AS A35, and AS A50.

2. Server workloads

The application domain for commercial server computers is diversifying at a significant rate. Recent developments such as the World Wide Web, the proliferation of collaborative groupware, and the introduction of new platform-independent languages such as Java** are forcing new requirements on computer systems in order to run these workloads effectively. At the same time, customer growth in the more traditional on-line transaction processing (OLTP), enterprise resource planning (ERP), and business intelligence/decision support applications drives requirements for increased performance in these domains as well.

Table 1 summarizes some key qualitative attributes of existing and emerging IBM server system workloads. The workloads include TPC-C [1], a widely used industry-standard benchmark that implements an OLTP database system for warehouse order entry; TPC-H [1], a decision-support benchmark that measures database query performance; NotesBench [2], which measures throughput for the Lotus Notes** collaborative groupware application; SAP two-tier [3], which implements an enterprise resource planning (ERP) application within the SAP R/3 environment; SAP 3-tier, which implements the same ERP application in a three-tiered environment that isolates the application code (tier 2) and database code (tier 3) to separate systems; and the Server Java application Java Business Object Benchmark (jBOB) [4], which is an OLTP benchmark implemented in Java.

Each workload is characterized in terms of seven attributes. The first is intra-thread parallelism, which describes the degree of instruction-level parallelism [5] that can be extracted from a single thread of execution in that workload. The second attribute is the instruction working set, which can be roughly characterized by the instruction reference miss rates observed in various levels of the cache. The third is the data working set, which can likewise be characterized by data reference miss rates in the cache hierarchy. The fourth is the degree of data sharing among threads, characterized by cache miss rates for read/write shared data. The fifth attribute is the degree of contention for shared data, which is characterized by the frequency of synchronization primitives such as load-reserve/store conditional [6].

Two attributes of workload I/O activity are listed next. The I/O request rate indicates the relative frequency of activity to the I/O subsystem directed by the application/operating system. Typically, these are disk accesses, or communications transfers over a LAN. The I/O data rate refers to the flow of data across system buses and interconnects between I/O and memory. Typically, this activity is measured in megabytes per second (MB/s). Generally, server workloads that stress

I/O with high request rates have relatively low data rates. These are usually cases of small-record access, requiring minimum-size data blocks to be moved to and from memory. By contrast, other workloads move large blocks of information, creating high data rates. However, there tend to be many fewer discrete requests to initiate the movement of data.

Clearly, there is a great deal of diversity in attributes over the range of server workloads shown in the table. Designing a server system to deliver robust performance across such a wide range of workloads without significant compromise in performance for any of them requires careful identification of the key parameters that must be optimized. Design guidance must be provided throughout the product development cycle to ensure that performance goals are met. Further complications arise from the fact that server workloads require complex, real-time interactions with the I/O subsystem, which in turn must interact with and react to asynchronous external events.

As an example of the complex factors involved in analyzing commercial server workloads, the case of the TPC-C workload can be considered. TPC-C presents many challenges to the system designer because of its low degree of intra-thread parallelism, its large instruction and data working sets, its pervasive and contentious data sharing, and its high rate of I/O activity. Furthermore, TPC-C receives considerable attention in the marketplace, and a competitive TPC-C rating provides a great deal of visibility. TPC-C is not just useful for benchmarking, however, since it stresses most important aspects of a server system, including the processor core, the multiprocessor interconnect, the memory hierarchy, and the I/O subsystem. As a result, TPC-C is widely used within IBM server development to identify design tradeoffs.

TPC-C consists of a set of five client transaction types that exercise a warehouse inventory database [1]. The service time for an individual transaction is dominated by I/O latency and requires relatively little CPU time. Hence, a balanced and fully loaded TPC-C system will have thousands of transactions in flight, with frequent processor interrupts and task switches driven by the initiation or termination of I/O requests. As a result, both the instruction and data working sets that the processor references are very large, and multiple levels of the cache hierarchy in a given system are referenced continually. Furthermore, the multiple in-flight transactions share data within the internal database constructs, and there is a great deal of database journal lock contention due to the ACID (atomicity, consistency, isolation, durability) requirements of TPC-C [1].

A set of key quantitative descriptors are needed to effectively characterize and model the performance of a server system running a complex workload such as TPC-C. These descriptors are typically broken down into processor core or on-chip attributes, memory subsystem or memory nest attributes, and system attributes for a given workload. Processor attributes include factors such as branch predictability, instruction mix, prevalence of dependence chains and load-compare-branch sequences, and frequency of shared-memory synchronization operations. These are usually characterized with deterministic trace-driven modeling and analysis, and are summarized with the on-chip or infinite-cache cycles-per-instruction (CPI) metric.

Another important workload attribute is its sensitivity to memory latency and the potential for memory accesses overlapping with other useful work. The intra-thread parallelism in server workloads such as TPC-C is meager and difficult to extract to an extent sufficient to successfully overlap off-chip memory latencies. Hence, server systems that implement multiple threads per chip (i.e., multithreading or multiple processor cores per chip) are best able to overlap long memory latencies with useful work, since there is plenty of parallelism available among the multiple threads.

The memory subsystem attributes of a server workload describe the degree of off-chip address and data traffic generated by the workload. These are typically quantified as miss rates, which specify the frequency of various types of misses and address transactions in the cache-coherent memory hierarchy. These miss rates are generated via a combination of hardware counters, special-purpose instrumentation, trace collection, and simulation. For TPC-C, miss rates tend toward the high end of the spectrum because of its nature of shared data manipulation.

Finally, the system attributes of a workload describe how the processor and memory subsystem communicate with the I/O subsystem. These are typically quantified as request rate and average request size for memory-mapped load and store references by the processor and frequency of interrupts and direct-memory accesses (DMA) by the I/O subsystem. Because of its somewhat random, record-oriented data-accessing algorithms from the file system, TPC-C tends to generate a relatively large number of small disk accesses, increasing the I/O request rate.

The other workloads listed in Table 1 all affect server design in varying ways that differ from TPC-C, but the concept of system interactions and performance optimization among several variables in the design is a constant. How this is accomplished is explained in the following sections.

3. Performance instrumentation

System instrumentation is the practice of adding hardware or software probes to a computing system for the purpose of monitoring the behavior of the instrumented system. When this data is used for performance-related activities,

the process is referred to as performance instrumentation [7]. The data collected by this process provides input for conducting analyses of existing systems and projecting the performance of future system designs.

Performance data collected from a live system provides designers with a means of judging how effectively various hardware features respond to typical processing patterns. This information can then be used to improve existing hardware or software configurations and guide future design decisions. Although the following discussion focuses on data-collection techniques for the purpose of making performance projections, it should be noted that the same techniques are used when gathering data for the analysis of existing systems.

Performance projections are typically based on software simulations of the proposed system. These simulations are driven by data collected from an appropriately instrumented system. Because the simulator output is directly affected by the performance data used as input, which in turn affects performance projections, tradeoffs, and optimizations, the extraction of high-quality performance data is key to any performance analysis effort. The quality of the performance data is a function of the instrumentation mechanism itself and the care taken in its deployment.

When instrumenting a system to supply performance data, it is imperative that the data collected be representative of the system's normal mode of operation. This ensures that performance improvements are targeted toward the system's most common set of operations. The workload used to drive the system under test therefore should produce realistic processing patterns, and measurements should be made only when this workload has reached its steady state. The method of instrumentation should also be designed so that it does not perturb the system under test [8]. For example, one direct approach to performance instrumentation is to augment a program with additional code that records the amount of time taken by various program events. However, the addition of this instrumentation code is also likely to change important indicators of system performance such as instruction path lengths, cache hit rates, and I/O utilization. If these perturbations are significant enough to be reflected in the data collected, subsequent analysis and projections based on this data may be erroneous.

The relative speed of the data-collection process must also be considered. Fast instrumentation enables the collection of greater amounts of data during a given measurement period, which in turn improves the statistical reliability of the data. In addition, a fast collection process is very desirable when the system under test is available for a limited amount of time. Although such time constraints would be considered minor in many environments, commercial workloads such as TPC-C often require systems costing several million dollars to run, making each benchmark test a precious resource not to be squandered.

Designing a performance instrumentation system that meets all of the above requirements is a challenge when doing studies of systems used for commercial processing. The complexity of commercial workloads and the increasing dimensions of the systems used to run these workloads tend to increase both the number of measurement points and the amount of data that must be collected from each point. The following section describes the performance instrumentation techniques used in the design of iSeries and pSeries systems and how these techniques address the above issues.

- *Trace collection*

One of the most direct means of monitoring the behavior of a system is to extract a trace of events occurring at a given point within the system. These traces can then be used to drive future software simulations, as discussed in Section 10. Many types of events may be traced, but typically traces of instructions executed by the processor, memory accesses, and I/O requests are of the greatest interest [9]. The method by which these traces are collected is a distinguishing characteristic in any trace-based projection effort. Because they share many common collection techniques, instruction and memory traces are described first.

Instruction and memory tracing

One of the simplest trace-collection techniques is to instrument an executable file with additional instructions such that the program itself produces a log of the path taken through the program at run time [10]. Postprocessing software then combines the contents of this log file with the instrumented binary file to produce a full instruction or memory reference trace for the program. However, this approach is typically applied to single executable files, making it inadequate for commercial workloads that consist of many user processes and tend to spend a significant amount of time executing operating system code.

An approach that allows for tracing multiple processes (including the operating system) uses programmable hardware within the processor to generate an interrupt whenever a user-specified event occurs [11]. The interrupt handler then writes the relevant trace information to a memory buffer, which is later written to disk. These events and handler routines can be specified in such a way that full instruction or memory traces can be produced for each process in the system. CTrace, an internal IBM tool for the pSeries server, uses this basic approach to generate traces.

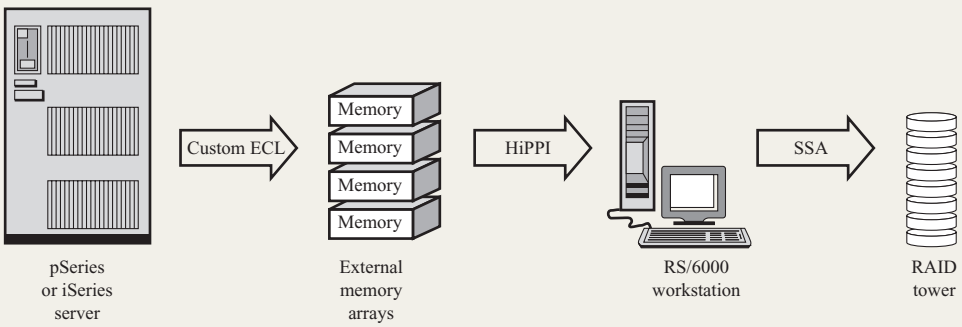

Figure 2

Common trace collection system for pSeries and iSeries servers.

Although CTrace is able to produce relatively complete trace information, tools such as this rely on frequent processor interrupts, which can perturb the system under study. To eliminate or reduce this effect, specialized measurement hardware can be used to extract traces by directly probing the system hardware. One of the more common techniques for hardware trace collection is to place snooping hardware on the main store bus, which monitors bus transactions and stores them to a large memory buffer that can later be written to disk. In this way, a trace of memory requests can be captured and used to drive later memory system simulations. The presence of processor caches means that only those memory operations that miss in the processor cache hierarchy are seen by the bus-snooping hardware. Although this has its advantages when one wishes to filter a trace [12], there are many instances (such as L1 cache simulations) for which a full address trace is needed.

The processor may also provide signals that can be probed to determine the instructions currently being executed. Because the collection of such an instruction trace must be done at the speed of the processor, the hardware for instruction tracing is more challenging to develop than that required for bus tracing. This effort can be justified by the wealth of information that can be derived from instruction traces. Not only is it possible to determine the processor's execution path with an instruction trace, it is also a means of collecting an even more complete memory trace than is possible with bus tracing.

For IBM servers, a common collection system is used to collect both processor and memory traces, as shown in **Figure 2**. Hardware probes are connected to a debug port provided on a specially designed processor board. This port operates at the speed of the processor and provides both processor instruction usage information and system bus snooping. This interface is connected to external memory arrays via a custom, high-speed ECL connection. In the current generation of the tracing system, these memory arrays have a total capacity of 16 GB. Once these arrays are filled, the trace data is downloaded via a HiPPI (high-performance parallel interface) connection to a pSeries workstation which processes the trace and stores it to an attached RAID (redundant array of independent disks) tower.

This collection system is designed to gather large traces as rapidly as possible, because there is typically a limited amount of time in which the workload driving the system remains in steady state. Multiple traces are collected for a single workload to ensure that nonrepresentative traces can be isolated by using standard validation techniques. Examples of nonrepresentative traces include those that happen to be taken while the operating system is involved in a large amount of paging activity or while a database journal is being written to disk. Although such events can have noticeable effects on performance, they are better addressed as special cases rather than being included in the analysis of the workload's steady-state behavior. When attention is focused on the traces taken during steady state, the performance optimizations derived from these traces can be applied to the most common workload operations and thereby have a greater overall effect on system performance.

Disk tracing

Unlike instruction and memory tracing, traces of disk accesses can be collected via software instrumentation without loss of information or undue perturbation of the system under study. Because I/O is managed by the operating system, it is possible to instrument a small

amount of code within the OS to keep track of all disk accesses. Furthermore, the effect of this instrumentation on overall system performance is negligible, because the extra processing time required to keep track of disk requests is small relative to the service times associated with these requests.

Software trace collection of the iSeries disk activity is done with the Performance Explorer tool. This tool activates instrumentation code within the OS to collect disk traces that can include such information as the type of operation (read or write), the number of sectors accessed, the destination disk unit, and the requested address.

- *Performance monitor counters (PMCs)*

Trace-driven simulations are capable of providing performance analysts with very detailed information on the behavior of a proposed system working under a given workload. This level of detail is not always necessary nor expedient to gather. PMC data can be collected and processed more quickly. This data is often used to analyze performance problems in hardware and software during the system optimization phase described in Section 10; it is also used to characterize new workloads without all of the work of collecting traces. If traces are collected, the PMC data can be used to select the most representative trace.

Processor PMCs

The current generation of PowerPC processors contain a set of PMCs that are dedicated to measuring particular events within the processor and its cache hierarchy. Each of the counters can be used to measure any of several types of performance metrics. These metrics can be divided into three basic categories:

1. *Event counters* Incremented whenever a specified event occurs—for example, the number of cache misses or correctly predicted branches.
2. *Event timers* Measure the total number of cycles taken to complete a given class of tasks. For example, the total number of stall cycles due to instructions accessing memory.
3. *Mask counters* Incremented whenever a user-supplied mask matches the contents of a completed instruction register. This is useful for deriving instruction frequency data for a particular instruction or class of instructions.

The data accumulated by these counters can be combined to derive a wide range of event probabilities and average cycle times.

I/O PMCs

The I/O adapters used in iSeries and pSeries systems also contain counters that count particular events or cycles. The statistics of most interest include average response times seen by the operating system, utilizations for storage adapters and the disk devices, read/write percentages, disk seek-distance distributions, and hit rates for the caches and buffers included in the storage subsystem.

- *Hardware emulation*

While performance counters are useful in summarizing system activity, they are limited to establishing the frequency and average latencies of particular events in an existing system. Conversely, traces can provide very detailed information for subsequent software simulations, but the traces themselves can be problematic. On one hand, it is desirable to gather as long a trace as possible to provide a statistically significant sample of trace events. On the other hand, the volume of data needed to model some configurations can be very large, making the trace files cumbersome to store and process.

Hardware emulation [13] provides a means of sampling large amounts of data without the need to store the sample in a trace file. In this approach, hardware probes are connected to a live system to capture trace events that are then used to drive a hardware emulator. This emulator replaces the function of a software simulator by mimicking the behavior of a proposed system design point. At the same time, the emulator collects statistics on the behavior of the emulated system.

Caching structures are particularly appropriate candidates for hardware emulation. Given the increasingly significant effect of memory latency on the overall performance of a system, designers have begun to rely on larger and more complex cache hierarchies to avoid long memory latencies. This trend presents a challenge to performance analysts because modeling very large caches using software simulation requires proportionally large traces.

Recognizing the limitations of trace-based cache studies, members of the IBM Server and Research divisions have co-developed a hardware cache emulation system [14]. The cache emulator attaches to the main store bus and is therefore able to snoop all memory transactions coming from the processor cache hierarchies. The emulator contains the logic and memory necessary to mimic the contents of up to four cache directories. Because the emulator design is based on field-programmable gate arrays (FPGAs), the board can be quickly reprogrammed to emulate a variety of cache configurations.

The cache geometries to be emulated are specified by the user via a PC console attached to the emulator card. The console sends configuration information to the card, which then updates the FPGA code for each directory to

set the emulated cache size, line size, associativity, and other attributes. The overall topology of the emulated caches is also specified by the user. For example, given a four-processor host machine, the emulator can be configured as a single cache shared by four processors, two coherent caches each sharing two processors apiece, or four coherent caches, each under a single processor. Once configured, the emulator card receives a signal from the console to begin the measurement for a fixed period of time. When the measurement completes, the contents of the emulator's statistical counters are downloaded to the console and stored to disk for later analysis.

One of the primary advantages of hardware cache emulation over trace-based techniques is the emulator's ability to measure cache behavior over relatively long periods of time. While most hardware trace collection systems are capable of sampling only a few seconds of a workload, the emulator's 40-bit counters allow data to be collected over several minutes or hours. It is this attribute of the cache emulator that enables it to model cache sizes that are too large for accurate trace-based simulation. Long sample periods are also desirable when the workload under study exhibits uneven cache behavior. In this case, the cache emulator can take measurements over the entire workload and therefore provide statistics on average cache performance.

Like the other instrumentation mechanisms, hardware emulation has its limitations. Although it is programmable, the emulation hardware is not as flexible as pure software simulators, which can be rewritten to model virtually any proposed design point. In contrast to trace-based techniques, the flow of events into an emulator cannot easily be repeated, making direct comparisons between subtly different design points more challenging.

- *Performance instrumentation summary*

Honoring the many constraints imposed on performance instrumentation yields a diverse range of techniques. The choice of technique to use for a given type of analysis is based on an understanding of the limitations and strengths of each method of instrumentation. Once a method is chosen, care must be taken that it is implemented in such a way that data can be collected unobtrusively and efficiently during a representative portion of the workload.

4. Model input generation

The quality of performance projections is heavily dependent on the quality of the data used to drive the projection models. For the most part, cache performance projections are based on trace-driven cache simulations and hardware event counts collected from systems already implemented. Unless validation of the projection process is the goal, the system configuration from which characterization data is collected will be different from the system configuration for which performance projections are targeted. Because of these mismatches, most collected data and the associated first-level analysis results do not correctly reflect the attributes needed for an accurate performance projection. Hardware changes, software changes, and the limitations of the collection system itself can all lead to projection aberrations. This section presents an overview of data manipulations that have been used to compensate for attributes that do not match between the measured system and the target system.

- *Accounting for hardware differences*

The target system might support a greater number of processors than can be found in current systems. Trend analysis is used to make up for a lower number of processors in the collection system. Detailed performance data (PMCs and traces) is gathered from current systems while varying the number of processors. Counter data and trace simulation results are then fitted to curves and combined. Extrapolations are made utilizing those curves for projecting cache component performance to match the number of processors in the target configuration.

Multithreading, like higher levels of multiprocessing, increases the number of concurrent threads in the system. Projections based on current systems may pose some difficulties. The target system may be multithreaded, whereas the current system is not. Even if the current system is already multithreaded, it may have implemented a considerably different multithreading design than the one being considered for implementation in the target system. In either case, detailed instruction and address traces of a single thread are split by task identification number and are fed to a pipeline simulator that emulates more than one concurrent thread.

In the multitasking environment found in commercial workloads, increased processor clock frequency leads to increased pressure in the supporting caches. Individual tasks, or groups of tasks, are dependent on the completion of I/O events for dispatch. Tasks face displacement of their working sets while waiting. Because I/O latency reductions generally lag behind processor clock rate increases, the risk of displacement is higher for faster processors than if the processors were running more slowly. Adjustments due to the multiprogramming level are based on measurements in which the clock rate is varied and a trend curve is derived.

- *Accounting for software differences*

Software, including both the operating system and applications, presents a difficult problem for projection. The first (and biggest) assumption is that the workload of today will resemble the workload of the target release date. This approximation can be made with more confidence if there is knowledge of how the software

and workload will evolve. Component attributes may be adjusted to account for software trends or programming commitments.

One example of compensation anticipates the effects of compiler optimization. Early in the development cycle, the modules that make up the operating system, database, and application have not been profiled [15]. Projections based on trace or PMC data from the early builds are missing the cache-friendly arrangement of code to be found in the final release. Projected miss rates are adjusted to reflect reductions in instruction cache and TLB miss rates when the system software has been profiled.

As another example, cache miss rate components may be adjusted to reflect the expected results of software engineering efforts aimed at reducing data sharing among processors [16].

Task interaction will affect miss rates. Compensation may be needed to account for contention over serialized code paths (such as spinning for a shared lock). As the number of processors increases, the amount of spinning increases, and miss rate projections may have to be modified accordingly.

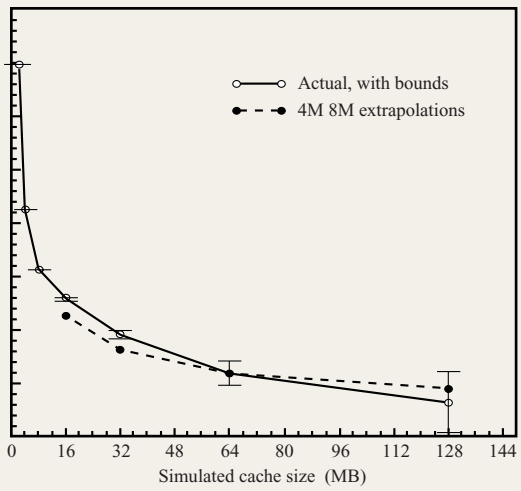

Figure 3

Correlation between extrapolated and actual cache miss rates.

• *Accounting for collection mechanisms*

The collection method and equipment itself may introduce artifacts that require compensation. Limitations in trace collection capacity may result in a trace that, in a simulated directory, produces a comparatively large number of references to uninitialized congruence sets. It is uncertain whether a reference to an uninitialized set should be considered a hit or a miss in the target system. A bound on this uncertainty produces a delta which is used to identify "good" simulation results. Only the simulation results from configurations with sufficient trace length (small deltas) are used in further calculations; those with wide deltas are discarded. Curves are fitted to the remaining points and are then used to project miss rates for those caches too large for individual traces. **Figure 3** shows an example of this process. The actual miss rate (obtained with longer traces) is shown as the solid line; the brackets define the bounds derived from analyzing shorter traces. If only the short traces were available, the 4M and 8M points would be selected for extrapolation to larger cache sizes. The dashed line shows the result of applying a simple log–log extrapolation to the 4M and 8M points.

Collecting bus traces from a multiprocessor system is a problem of balance. If the bus traces are collected with no underlying caches, the system runs too slowly to provide a worthwhile representation of memory and cache interactions. If too large a cache is used as a filter, the ability to simulate the smaller-sized caches is lost.

Assuming that balance can be obtained by using a small direct-mapped cache as a filter, it still may not be possible to consider a memory component in isolation from the hardware configuration. For example, the miss rate of a second-level cache may depend on the first-level subsystems that interact underneath that cache. (A large TLB puts less pressure on a data cache than would a small one.) If the configuration of the collection system does not match that of the target system, adjustments are made to the simulation results. These adjustments are based on simulating both the current and the target directory configurations with instruction-address (unfiltered) traces and then applying a relative adjustment factor to bus (cache-filtered) trace simulations.

• *Accounting for the collection system*

Commercial benchmark setups tend to consume large quantities of resources, first to tune and then to measure. These resources include operators, performance analysts, disk arrays, system memory cards, software support, driver systems, and floor space. For systems large enough to serve as worthwhile measurement platforms, time slots available to performance data collection are usually limited in frequency and duration. Measurement opportunities represent a compromise between what is wanted for projection work and what resources can be spared from development. The situation can cause missing measurement points, making trend analysis more difficult.

Even if there are sufficient resources to collect data for projection purposes, there may be other reasons why the collected data may not be suitable for direct use. Not all configurations in a series may be suitable for use in direct

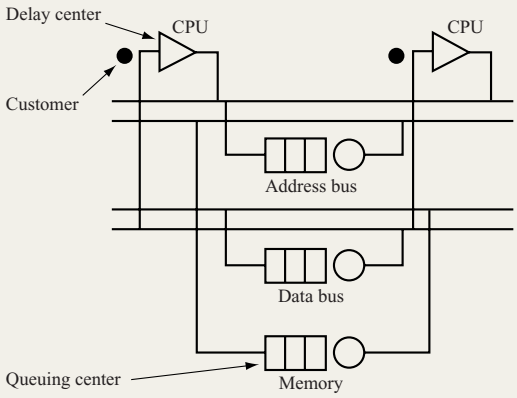

Figure 4

Closed queuing network used to construct a simple model of a two-processor system.

comparisons because of variations in workload setups. Great care must be exercised to ensure that series of benchmark runs yield comparable data. Disk balancing, number of simulated users, size of memory pool, and utilization are all key attributes that must be monitored [17].

To make the data collected under such conditions relevant, adjustments of measured components may be needed. Component adjustments are performed with a combination of curve-fitting tools of both the automatic and manual varieties.

5. High-level models

There are several models and simulators that are driven by the data described in the previous section; one such set of models is known as high-level models. These are models that are written early in the design process to make gross tradeoffs among alternative cache structures, bus bandwidths, and topologies. A wide variety of alternatives are typically considered, so high-level models must be easy to write and modify, and should have a short execution time. Systems at this level of detail can usually be represented as closed queuing networks of infinite queues, which can be described and analyzed quickly using analytic methods. This section describes how these queuing models are written and solved.

In practice, because of their speed and flexibility, high-level models are used over the entire design cycle from concept to customer delivery. This is because a family of servers includes several tens of configurations, each of which requires a performance projection for several workloads. Furthermore, configurations can vary widely until late in the design cycle, requiring frequent modeling runs. It is less time-consuming to use a high-level model rather than a trace-driven model to maintain projections on so many variations.

- *Mean value analysis*

A closed queuing network (**Figure 4**) is a collection of interconnected queuing centers with a fixed number of customers (shown as tokens) that cannot leave the network. Each customer has an initial position in the network from which it can travel along one or more randomly selected routes, and each route must return to the initial position. At each visit to a queuing center along a route, a service time distribution is specified for that center. When a customer leaves a queue, it may randomly select one of several destination queues.

Mean value analysis (MVA) is the name of an exact algorithm for finding the mean performance measures of "product-form" closed queuing networks [18, 19]. A closed queuing network has a product-form solution if the customer routes satisfy certain properties:

- The service distributions must belong to a narrow class, the most common being the exponential distribution.
- The selection of a queuing center at each step in a route must be independent of all previous routing selections (Markovian routing).

The mean value analysis algorithm is efficient, so complex networks with hundreds of queues and hundreds of complex routes can be solved in seconds.

Closed queuing networks are a natural representation for computer systems at a high level of abstraction: Queuing centers represent microarchitectural resources, and customers represent cache misses. Examples of microarchitectural resources include cache reload logic, buses, memory arrays, and a processor as it behaves with infinite level-one caches. While the network topology and service times are determined by hardware assumptions, the routing probabilities are determined by workload characteristics, such as miss rate.

Mean value analysis is the preferred method for analyzing closed queuing networks because it takes into account the finite number of customers (i.e., misses) in the system when queuing delays are computed. There are *ad hoc* approaches for analyzing closed queuing networks that use open queuing network delay equations (based on results for M/M/1 queues). These are pessimistic, however, in that they can result in queuing delays that would require a much larger number of outstanding misses than is possible. This problem is most acute when a system has one or more resources that are at or near saturation.

Queueing network models need not be solved analytically; they can also be solved via simulation.

Analytic solutions are preferable, however, since they are much faster, and many studies have shown that analytic approximations are quite good [20]. Simulation is a good alternative when the limitations of MVA are expected to have a significant impact on performance. An example of this would be some dominant finite queue behavior. Sections 7 and 8 describe the use of queuing network simulation to model important finite queue effects in I/O subsystems.

• *EZMVA: A generic MVA solver optimized for computer systems analysis*

Despite its computational efficiency, the equations for a complex MVA model with many processors and routes are cumbersome to construct. It is common for an MVA model to use hundreds of queues with hundreds of hardware and software parameters. This results in a thousand or more MVA equations. IBM has written a queuing network analyzer, called EZMVA, that automates the creation of these equations by compiling them automatically from an abstract description of the system.

An EZMVA model description contains three parts:

- A parameter section, where hardware and workload parameters are declared.
- A topology section, where queues are defined.
- A routing section, where customer routing (timing) is defined in a natural way as a list of visits to queuing resources.

A model description is usually highly parameterized and represents a family of configurations with different numbers of processors, optional caches, variable latencies, and adjustable bus attributes.

The compiled model is only a single piece of the EZMVA modeling environment. The complete environment includes a source of workload and configuration parameters, a model solver, and a variety of output statistics and diagrams.

• *Limitations of mean value analysis*

Some characteristics of computer systems violate the requirements of product-form closed queuing networks:

- Most computer systems use discrete time instead of continuous time. In practice, this has little effect on accuracy.
- Few resources exhibit exponential service times. Most resources exhibit deterministic (or constant) service times. An approximation technique is used to model deterministic service times [21]. This technique has been empirically shown to be accurate.
- Some machine resources use a non-FIFO queuing discipline, and the actual queuing discipline can be quite complicated. In practice, the FIFO discipline is often sufficiently accurate. It is possible to use approximation techniques for some types of priority queues.
- No machine resource has infinite queue space. Even if finite queues could be modeled, there are widely varying schemes for handling queue overflow that require complex interactions among groups of queues. In the early stages of system design, it is usually assumed that queues will be sized so that queue overflows will be infrequent. When finite queue effects must be identified or modeled, a more detailed simulation model must be used.
- Processors must either stall or not stall on a cache miss; overlap scenarios cannot be modeled. This restriction has not been a problem for level-one cache misses until the most recent generation of PowerPC processors. These have sufficient out-of-order resources that there is significant overlap between level-one misses that hit in the level-two cache. Fortunately, little overlap occurs for level-two cache misses, so the processor can be represented as a delay center with service time equal to the average interval between level-two cache misses.

6. Detailed timer models

• *Overview of timing simulator*

To carry out detailed performance studies, a cycle-accurate model of the system is written. This allows for detailed analysis of the workings of the design as it evolves. The goal of the model is to do detailed design tradeoffs when looking at low-level interactions between instructions and events in the system. The MVA model, described above, covers the high-level memory-system tradeoffs but does not cover tradeoffs in the processor core, or more detailed tradeoffs in the memory system. A model that does this is generally referred to as a "timing simulator." Examples of modeled details include number of functional units, pipeline depth, thread-switch algorithms, cache-replacement algorithms, cache snooping, and DRAM bus utilization. The timing simulator covers core pipeline details, caches, TLBs, and the memory system of a multiprocessor system. Inputs consist of instruction traces and other workload characteristics described by probabilities. In general, the timing simulator described here is similar to various other timing simulators [22–24]. Significant features are the use of probabilities to augment trace-driven simulation and the use of the simulator in multiprocessor configurations. This section describes the scope of the simulator and some of the methodology used to evaluate performance.

A model can be constructed in several different ways, and different degrees of detail can be implemented. Some

different approaches to modeling are the following: 1) a very-high-level trace-driven simulator to determine gross tradeoffs; 2) a detailed cycle-accurate trace-driven simulator, either fully deterministic, fully probabilistic, or a combination of the two; 3) a detailed cycle-accurate execution-based simulator; and 4) actual logic design (represented in a hardware description language, or HDL) used as a timing simulator for performance analysis. The approach taken for processor and memory system modeling is the second one, with some aspects of the first included. Initially, the model is developed at a high level in order to compare more gross design decisions. Since there is an MVA model (described above) to do high-level analysis of the memory and MP system, the high-level timing simulator, described here, is not used for these tradeoffs. However, a simple memory model is used to introduce cache-miss effects into the analysis, including effects on the processor core. This type of simulator covers design options such as the number of execution units and their function split, pipeline lengths, and branch-prediction algorithms. As the design progresses, more details are defined, and the timing simulator becomes more specific. The memory system also becomes more detailed as memory tradeoffs for the system (including multiprocessor topology) are made using the timing simulator. The timing simulator and MVA model are compared to ensure that they correlate well.

The choice of a trace-driven methodology or an execution-driven methodology is based on several constraints, as described in Section 3. Given the nature of commercial workloads, it should be clear that running in an execution mode requires the simulation of much more than a single processor. The size of the system to simulate becomes prohibitively large, and the interactions among operating system, application, multiple processors, and I/O become very complex. Because traces are available, they provide a primary source of inputs to the timing simulator. Therefore, a trace-driven methodology is used rather than an execution-based methodology.

The trace-driven methodology has its limitations, however. To overcome these limitations, as described above, additional timing simulator inputs consist of probabilities of events that are not readily available from the traces, are in error on the traces, or are expected to change in the future. Because traces are collected from a current system, and the system for which performance tradeoff analysis is done will be shipped some number of years later, the collected traces reflect an old system by the time the new one ships. Building systems with more processors than previous systems, or with multithreading, are primary examples of the change in workload characteristics over time. In addition, there will be new workloads, for which there are no traces yet, that are expected to become more important by the time the new system is available. Since the exact characteristics of the future workloads cannot be known, the use of probabilities provides a method to vary the workload characteristics to observe behavior for a range of such characteristics.

When doing performance tradeoffs for a multiprocessor system, running completely from traces is difficult and misleading. The solution used during performance modeling for the systems described in this paper is to do uniprocessor core tradeoffs using uniprocessor traces. The uniprocessor part of the timing simulator has the capability to do a pure trace-driven simulation in addition to using probabilities. The trace is used as input for providing a stream of instructions and inter-instruction register and memory dependencies. When a cache access is initiated, a probability or a cache directory model can be used to determine hit or miss. Note that addresses on the trace are still used for memory dependencies, including different references to the same line or page, when running with probabilities. This allows simulation of load-store interaction, unaligned data, and multiple references to a line that has a cache miss. In addition to supplementing the traces with probabilities, traces are modified to reflect any important changes that may occur at the instruction level, such as new instructions added to the architecture.

From the multiprocessor standpoint, probabilities are the key timing simulator input, since complete MP instruction traces are not available, even for the number of processors in current systems. The multiprocessor characteristics are summarized in a set of probabilities describing where and in what state different types of requested data can be found. For example, given an L2 cache miss, data may be found modified in another processor's L2 cache, found shared in one or more other L2 caches, or not found in any L2 cache, in which case the data comes from main memory. The probability specifies which of these instances is to be used, which processor supplies the data, if any, and which caches must invalidate lines. The act of snooping is performed in the timing simulator by modeling buses, directories, etc.

Many different types of events can, in general, be modeled deterministically or probabilistically. Probability modeling can begin with simple random events, given an exponentially distributed probability; this is what is usually used when running the model. However, other distributions can also be used. Specifically, most events exhibit more clumping than is described by an exponential distribution. For that reason, the model also implements hyperexponential distributions, which we have found to produce better models of event arrivals in real systems. When running the model, the user can specify the characteristics of the probabilities. A second class of improvements is correlation of events. In our models, correlation effects are limited to a combination of a single

probability and subsequent address comparisons (e.g., two references to the same line).

- *Modeling process*

The full model is constructed in three parts: core model, cache and translation model, and memory nest model. The core model consists of a detailed uniprocessor model covering the pipeline structure from instruction fetch through execution and completion. This part does not include cache or translation. When there is a memory request, the cache/translation part of the timing simulator is accessed to determine a hit or miss, as well as to simulate the details in the cache/translation pipelines and buffers. The hit or miss determination is made using a deterministic cache directory or a probabilistic cache directory. This part includes the L2 cache, the primary cache for multiprocessor coherence, and the TLBs. The third part couples multiple processors with a detailed model of the memory system. In a multiprocessor system, only one of these detailed pipeline models is used. The remaining processors are characterized by a very simple processor model, in which infinite-cache CPI is a primary input. These processors access the same cache/translation models and are used primarily as traffic generators for the system. The timing simulator therefore consists of one (or zero) detailed instruction-level timing simulators and $N - 1$ (or N) simplified processor models. Each of the N processors has its own private caches (as appropriate for the design) and any shared caches and memory that are in the system.

The modeling process begins as the next design is contemplated and the high-level direction is starting to take shape. By this point, there is an analytical model used to help define the overall structure of the system and some high-level pipeline model in place to cover that part of the system. The model writer is writing to a specification that is derived primarily from verbal descriptions from the design team, with some block diagrams and timing diagrams. The high-level design evolves in part from feedback from performance information from the high-level models. As the design progresses, more details are determined, and those are implemented in the timing simulator. Details include the facilities that are available and the expected timing relationships expressed through timing diagrams. The process begins rather informally but becomes more formal as the design is documented.

The most important part of this process is the evaluation of design tradeoffs. The design team must evaluate the performance tradeoffs of alternative designs. Most of these performance tradeoffs are made using the timing simulator to compare the designs when running with the traces and probabilities described above. In addition, outputs from the timing simulator are used to identify problem areas in the design or areas where the design can be scaled back without hurting performance. An important part of the process is the generation of timing diagrams by the timing simulator. This provides feedback to the design team on the behavior of the design and provides an indication of how well the timing simulator is matching the designer's description of the design.

Another significant use of the model is for system performance projections used for positioning the product and defining different models (cycle time, external cache size, number of processors, and other configurations). This process is similar to that used for design tradeoffs and also takes into account software path length and I/O effects; I/O traffic is included in the model and is specified by a rate (requests per completed instruction), block sizes, and other characteristics. The timing simulator is used for calibrating the high-level MVA model as well.

The timing simulator is used primarily for running a trace augmented with probabilities derived from workload characterizations; in addition, the use of probabilities offers another advantage. This area can be characterized as sensitivity studies. In these studies, workload characteristics can be altered to represent characteristics of new workloads, if there is some information available to describe them. Also, the existing workload characterizations are altered in order to determine the stability of the design. This helps to make the design more robust over a larger variety of workloads.

After the high-level design is completed and much of the logic design is completed, the HDL simulator enters the performance picture as a means of verifying the timing simulator against the actual design. At this point, the timing simulator can be compared to the HDL simulator for performance verification. A series of short test cases are written; each tests one or two specific parts of the design. The shortest test cases may be only three or four instructions. A goal is to keep the number of test cases to a manageable level. As a result, the situations deemed most important for performance are chosen as test cases. One or two hundred test cases is enough to satisfy these goals, although the number will be a function of the complexity of the design and the range of workloads of interest. The test cases are run through the timing simulator and the hardware description language model and are compared both for total number of cycles and for activities on each cycle in the test. As differences are noted and understood, either the timing simulator or the logic design is changed. For subsequent versions of the hardware design, the test cases are also run so that the model accurately reflects the final hardware design.

The timing simulator is used after the first-pass hardware primarily for analyzing the effect of subsequent changes due to bugs discovered during functional

verification and testing or to design changes to improve performance (CPI or cycle time), and for software optimization, including compiler tuning for the processor. In the latter case, the timing simulator is made available to compiler developers who can optimize the code to the specific processor design.

7. I/O devices and adapters

In order for server workloads to take full advantage of the performance capabilities of the processor and memory subsystem, the I/O subsystem must be sized to handle the I/O traffic generated by the workload. The I/O interconnect, which connects the I/O adapters and devices to the memory subsystem, must have sufficient bandwidth for all of the disk, tape, and communications data and control information that passes between the I/O adapters and memory. The storage subsystem, consisting of the storage adapters and the disk and tape devices, must be able to handle the traffic generated by the workload while minimizing the response time. Designing a server whose I/O subsystems properly match the performance capabilities of its processor is not a trivial task. The system designers and planners must balance the cost of the I/O components against the requirements of the workloads, and must ensure that no I/O component becomes a performance bottleneck. Performance analysis plays many roles in this process. In the remainder of this section, we discuss how performance analysis is used during the design of the storage subsystem of server systems. The use of performance analysis in the design of the I/O interconnect is discussed in the following section.

For personal computers and single-user workstations, the key criterion in selecting storage devices and adapters is typically capacity, with performance being a secondary criterion. For servers, the performance of the storage subsystem is critical; capacity, while important, is secondary. OLTP workloads generate a high throughput (operations per second) to the storage subsystem. Business intelligence workloads require high bandwidth (megabytes per second) for data flowing from the storage subsystem. If the storage subsystem is not designed to provide adequate performance, it becomes a bottleneck which prevents full utilization of the capabilities of the system. If the storage subsystem is designed to meet the performance requirements, the capacity requirements can usually be satisfied as well.

Performance analysis of the storage subsystem starts with the characterization of the I/O properties of the server workloads. These characteristics are discussed in Section 2. This information is used to establish the performance requirements for each component in the subsystem. These requirements are stated in terms of the operations per second (ops/s) that the subsystem must be able to handle with a specified acceptable response time, and the peak megabytes per second (MB/s) that must flow from each component. For the storage subsystem, the ops/s requirements are determined on the basis of the characteristics of the OLTP workloads. The ops/s requirement is the number of operations per second that the subsystem must handle at a specific average subsystem utilization, with a specified maximum response time. For example, for iSeries systems, this requirement is specified at an average utilization, over all of the disks in the subsystem, of 40%. This value was selected in order to ensure that the utilization of the most heavily used disk is no greater than 60–70%, beyond which response time quickly becomes unacceptable. The MB/s requirement is determined on the basis of the characteristics of business intelligence and other data-intensive workloads. This is the amount of data that must be able to flow from the device buses into the memory subsystem.

More detailed characterization of the I/O workload is then performed in order to obtain the parameters needed in the detailed analysis of the I/O components. The workload characteristics are used to drive the detailed performance analysis during the design phase of each component. Performance analysis during the design phase serves two purposes: maximizing the performance of the components and developing projections of actual performance. By maximizing the performance of each of the I/O components, we can build balanced system configurations with a minimum number of components. By projecting the performance capabilities, we can ensure that selected I/O configurations meet the I/O requirements dictated by the selected workloads.

The storage subsystem and its relationship to the rest of the system is shown in **Figure 5**. The main components of the storage subsystem are the storage adapters, the disk devices, the removable media devices, the adapter bus that connects the adapter to the I/O interconnection, and the device bus that connects the various devices to the adapter. Each of these components possesses characteristics which affect the overall performance of the subsystem. High-function storage adapters used on most server systems include powerful microprocessors, complex microcode, and special-purpose hardware. These are used to manage interactions with the I/O interconnect, perform DMAs of data and control information, manage protection schemes (such as RAID [25] or mirroring) on arrays of disk devices, and implement protocols (such as SCSI-3) used to interact with the storage devices. In addition, the storage adapters often contain memory dedicated to a nonvolatile write cache, which is used to reduce response time on write operations and to reduce the number of operations actually occurring on the disks, and a read cache, which is used to reduce the response time on read operations. Management of these caches is also done by microcode running on the local microprocessor. The most

important performance characteristics of the disk devices are seek time, rotational latency, and command latency. In addition, disks have differing amounts of read-ahead buffer memory. The most obvious effect of the buses on the subsystem performance is on the maximum bandwidth that can be achieved. However, the transfer latency of the bus and the impact of bus arbitration schemes also play a role in the overall performance.

The performance analysis of storage subsystems is done in a hierarchical fashion. A high-level analytical queuing model, as described in Section 5, is used to compute both system- and subsystem-level performance effects of changes in the storage subsystem. The model takes as parameters the characteristics of disks (e.g., seek and rotational latencies), buses (e.g., peak usable bandwidth) adapters (e.g., service times, RAID overheads, and write cache size and type), and workload, including the impact of the workload on such subsystem parameters as write-cache effectiveness (disk writes to nearby sectors may be combined, reducing the number of writes flowing to the disks), read-ahead buffer hit rate, and seek-distance distribution. The outputs of this model are projections of system-level throughput, throughput vs. response-time curves, and component utilizations for the I/O subsystem. This type of model allows calibration with existing designs, projection of future changes, and analysis of sensitivity parameters. The parameters for this model are obtained from a number of sources, such as detailed modeling of components, analysis of traces, measurements, and published device specifications.

Detailed performance modeling is typically used to evaluate the performance of the storage adapter and to understand the performance effects of interactions between the adapter and disks. Both analytical and simulation models are used. Analytical models are typically used to project the impact of changes in existing designs on performance. For example, analytical models are used when projecting the performance impact of changes in processor, cache structure, memory controller, or memory technology on overall adapter performance. Creating these models requires analysis of address traces and measurements of the current design in order to determine the percentage of the overall latency which is attributable to the various sources (e.g., in-core computation, satisfying cache misses, processor loads, and stores to device control registers). These investigations are made significantly simpler than those for the main processor by the fact that the processor workload of a storage adapter is relatively static and can be well characterized by a few well-chosen address traces. Similar analyses can be performed to determine the effect of changes in microcode path length and data structures.

Simulation models are used to aid designers in making complex design tradeoffs and to determine additional

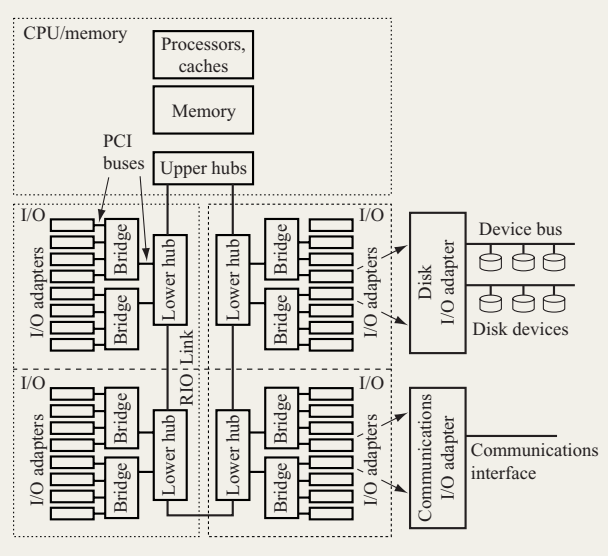

Figure 5

Model of the storage subsystem.

parameters for the high-level model. The simulation models are typically driven by traces of host-generated disk operations. The collection of these traces is discussed in Section 3. The traces are collected from benchmarks (e.g., TPC-C, TPC-H, NotesBench) and from customer systems running actual workloads. The traces used represent a large range of possible environments. In addition, traces are synthetically generated, and alterations are made to existing traces, in order to evaluate the effect of anticipated changes in future workloads as described in Section 4. For design decisions, these traces are used to drive a simulation model of the storage subsystem. This model is written in C++ using the CSIM [26] modeling libraries and includes details about the storage adapter, device bus, and disks. Each component is parameterized to allow its characteristics to be altered to represent a wide range of actual hardware. This simulator has been used to examine the effect of changes in adapter buffer sizes and structure, disk technology, and RAID parity layout on subsystem-level performance. It can also be used to determine seek-time distributions for different workloads. It is similar to, but less general than, the Disksim Simulator [27], since it has been tailored to the specifics of IBM storage subsystems.

In addition to their use in driving the simulation models, the disk traces are analyzed to determine various workload-specific parameters. For example, the traces are run through simulators of the adapter's write and read caches in order to determine hit rates and write-cache

efficiencies. The write-cache efficiency is the percentage of write operations generated by the system that are later issued to a disk.

As both a validation tool and a source of model parameters, measurements are a key part of the performance evaluation of storage subsystems. Measurements of the throughput, response time, and utilization for existing designs are used to validate the high-level performance models. Measurements of detailed parameters, such as write-cache efficiency, can be used to validate the detailed simulators. Measurements of existing designs greatly simplify the modeling of proposed design changes. In addition, measurements of usable bus bandwidth are critical in determining whether the subsystem will meet the MB/s requirements.

The end product of the performance analysis of a storage subsystem is a projection of the subsystem response time and utilization at a range of throughputs and for a range of workloads. These projections can be used to determine the number of adapters and disks needed to meet the requirements for a system under different workload conditions. In addition, by using performance analysis to guide key design decisions, we can optimize subsystem performance and therefore reduce the number of components required in the storage subsystem. This, together with performance analysis of the I/O interconnect and other I/O subsystems, can ensure that no bottlenecks exist in the I/O subsystem that prevent the main processor and memory subsystem from achieving its performance potential.

8. I/O interconnection performance

Large servers often attach hundreds, even thousands of disk drives to meet the demands of commercial workloads. Similarly, these servers attach many communications lines, both LANs (local-area networks) and WANs (wide-area networks). This requires that the I/O interconnect for large servers support the attachment of hundreds of I/O adapter cards that access the system's main memory at high I/O bandwidths.

IBM servers address these requirements using the structure shown in Figure 5. The I/O slots for attaching I/O are physically located at some distance (5 to 15 meters) in a box separate from the CPU and its memory. The upper hub shown in the figure resides close to the CPU/memory and generates several remote I/O (RIO) links. Each RIO link is a high-speed, byte-wide, full-duplex interface designed to handle a distance over copper wires of 5 to 15 meters. The RIO links connect to lower hubs, which in turn produce one or more PCI (peripheral component interconnect) buses. Together, the upper hub and the lower hub perform the function referred to in the PCI specification as the processor host bridge (PHB).

In addition to creating PCI buses, the lower hub has another set of RIO links that can connect to additional downstream lower hubs. The last lower hubs in a pair of RIO links are connected, creating a RIO loop. This loop provides an alternate path for I/O traffic in the event of a break in one of the RIO links. The "passthrough" function of the lower hub is designed to be low-latency. It forwards the packets received to the next link without waiting for the entire packet to be received. The PCI bus produced by the lower hub can have I/O attachment cards and bridges in accordance with the PCI specification.

The I/O adapters in Figure 5 are standard PCI bus adapters. Some are designed to meet the special needs of a commercial server; others are off-the-shelf PCI cards that are common in the industry.

- *I/O adapter distance to memory*

The arrangement described above solves the packaging problem of getting all of the adapter cards into the same box as the CPU and introduces the concept of a "distance to memory" to I/O. In client systems, where one or two PCI buses are sufficient, the distance to memory is small enough to be negligible. In large server systems, this distance can become a significant problem in maintaining bandwidth and controlling latency.

The performance goal for the lower hub and bridges is simply stated: Be able to sustain high bandwidths on the PCI buses while at a significant distance from memory. Exploring various approaches to the design of the upper hub, lower hub, and bridge, or changes to the RIO communications protocol, is the primary function of I/O interconnection performance modeling.

- *I/O interconnect simulator*

The performance simulator is written in C++ and uses CSIM [26] as its simulation engine. The simulator is cycle-accurate and has an object-oriented structure that closely parallels the structure of the physical components in Figure 5. The I/O traffic used to drive the simulator is synthetically generated, using probabilities, by objects representing the I/O adapters (IOA objects) shown in **Figure 6**. Each IOA object attempts to access its simulated PCI bus in a manner similar to the way in which a real IOA would attempt to access its real PCI bus. The simulated PCI bus can be allocated to just one simulated IOA at any moment in time, causing feedback to IOAs which fail to get the bus, delaying their requests. In this manner, contention for shared resources (arbiters, buses, buffers, RIO links, etc.) is resolved throughout the model.

Figure 6 shows a simplified view of the object structure of the simulator. The term "fabric" is used to describe collectively the upper hub, RIO links, and lower hub. In the simulator, fabric is not a class but a source file that comprises the upper hub, RIO, and lower hub classes. The

protocol on the RIO links resembles a communications-oriented protocol more than a bus-oriented protocol, with acknowledgments of the successful receipt of transmitted packets flowing on the opposite sides of the full-duplex links. The class definitions of the upper hub and the lower hub are dependent on each other. By putting both definitions in the fabric file, the interactions between the lower hub and the upper hub can be made clearer, reducing the number of errors in the code.

The simulator also includes the processor's accesses to the I/O adapters. The increase in latency caused by the need to locate the I/O buses at some distance from the CPU and memory causes the latency of the processor's memory-mapped I/O load instructions to increase. The increase can be an important factor in the performance of the processor. The purpose and behavior of the processor object are analogous to those of the IOA objects. Processor traffic is synthetically generated by each processor object as it contends for the resources in the upper hub. Note that there are many processes running simultaneously on processor objects, generating the I/O traffic.

From the model, we extract the I/O bandwidth per PCI bus, per lower hub, per RIO link, and per upper hub. We also gather information on the latency seen by the I/O adapter to first data and to last data. From the processor's viewpoint, we gather information showing the distribution of latencies for memory-mapped I/O stores and loads. We use these metrics with various design options to increase bandwidth, lower latency, and reduce cost or design risk of the components of the I/O interconnection structure.

The I/O interconnection simulator is cycle-accurate and focuses on the upper hub, lower hubs, bridges, RIO links, and PCI buses that make up the I/O interconnect. Its purpose is to determine the best implementation of each component as a functioning part of a whole. The workload for the simulator is synthetically generated by objects acting as processors generating I/O requests and handling responses and objects acting as I/O adapters doing DMA reads and writes to memory in response to the processor requests.

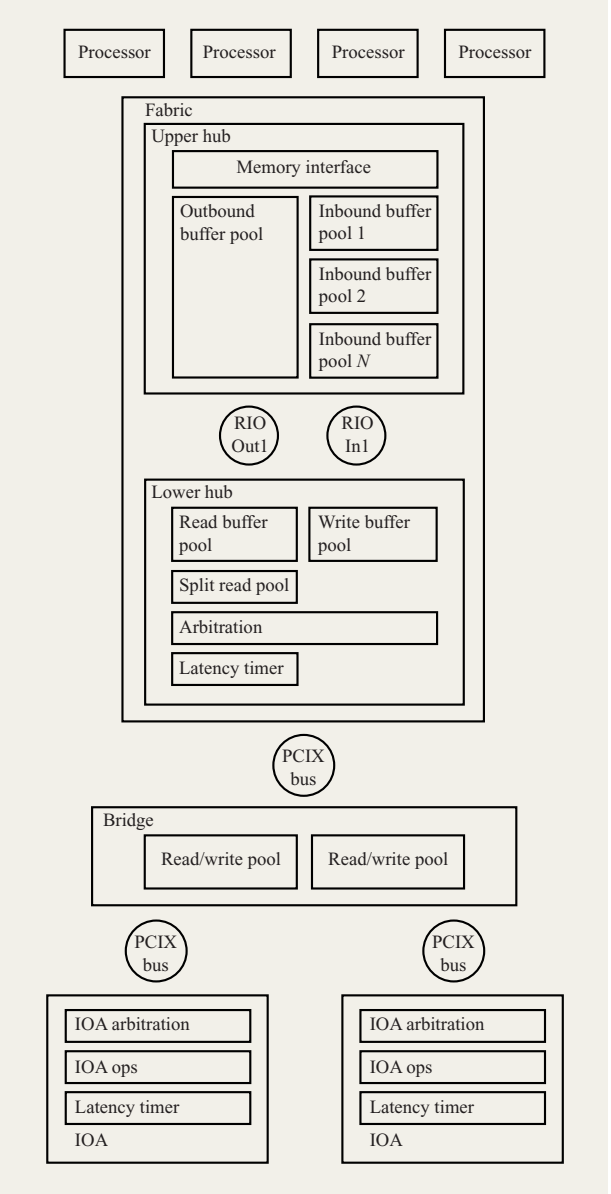

Figure 6

Simulator object structure.

9. An example commercial server

The above methodology and server workloads are used in the development of some PowerPC processors. As an example, a processor code named Northstar and its associated memory system are described. This processor is known as the A50 and also as the RS64 II. It became available for purchase in both product lines in the third quarter of 1998. A later version of this processor, SStar, known as the RS64 IV, began shipping in the third quarter of 2000. The RS64 IV is described in more detail in an accompanying paper in this issue of this journal [28].

Because this processor was used only in iSeries and pSeries servers and not in any workstations, it is optimized solely for server workloads. As a result, the microarchitecture of this processor is somewhat different from those of other processors. Because of the high cache-miss rates of server workloads, the processor spends a large portion of its execution time stalled on these misses, which reduces the benefit of a fast clock. Instead of using a fast clock which requires a long pipeline, the four-dispatch-wide, in-order pipeline is kept short and efficient. Characteristic of this approach, this processor optimizes specific

control paths. Server workloads tend to have more loads and stores, with a higher frequency of the load-use dependency. They also have a higher frequency of the compare-branch dependency. By keeping the pipeline short and optimizing these critical dependencies, it is not necessary to have dynamic branch prediction [28]. Server workloads also have a large instruction footprint, requiring large tables to obtain good branch-prediction rates. The high cache-miss rates and frequent dependencies limit the instruction-level parallelism of these workloads [29–31]. This limits the benefit of out-of-order execution and register renaming. These features also add significant chip area and design complexity. Instead of including these features, the chip area is used to address the cache-miss portion of the CPI.

The most unique feature included in this processor is multithreading. This feature was added specifically to address level-two cache misses, because they are a significant portion of the CPI and they are long-latency events which multithreading can cover well. Multithreading utilizes the natural, thread-level parallelism that exists in server workloads which are multiuser and multitasking. By utilizing another thread, useful processing can be performed during long-latency events such as level-two cache misses. Because these events comprise a significant portion of the execution time of a single thread, multithreading significantly increases the throughput of the processor. For further information on the implementation of multithreading in this processor, see the accompanying paper in this issue of this journal [28].

In addition to adding multithreading to address the stall time of long-latency, level-two cache misses, the significant CPI component of the level-one caches was addressed by making them large and making the level-one instruction cache line size 128 bytes. Both the instruction cache and data cache were 64 kilobytes. This reduced the miss rate and the portion of the CPI associated with level-one cache misses. The level-two cache is also large and associative; making it four-way associative significantly reduces the miss rate.

All modern servers are multiprocessors, and multiprocessors contain an additional source of misses. These misses come from the cache-to-cache movement of read-write shared data. When a level-two cache miss occurs, the command sent to the memory system is snooped by all processors. If one of the processors has the line and it has been modified, that processor must supply the line of data to the requesting processor. Since these misses do not occur in a uniprocessor, the miss rate of the level-two cache is higher in a multiprocessor. For some server workloads, these misses can be a very significant portion of the level-two cache misses [31]. They are also very important to the scalability of both hardware and software. Therefore, the cache-to-cache miss latency of this processor and its memory system is optimized. The result is a latency for a cache-to-cache movement that is significantly faster than accessing the main store.

10. System performance optimization

All of the previous sections have focused on performance analysis of hardware (processor, memory system, and I/O system). However, there is also a great deal of performance to be gained from optimizing software. Tuning compiler optimizations to the microarchitecture of the processor is widely recognized as beneficial. While this is necessary for server workloads, it is not sufficient. Server workloads are large and complex, requiring higher-level optimizations. The unpredictability of the memory-access patterns makes it difficult for the compiler to significantly affect the cache-miss rates. More significantly, the multiuser, multitasking nature of server workloads and the multiprocessor nature of the hardware lead to complex interactions among the cache footprints of the many tasks and the movement of read-write shared data among the processors. Optimizing the software with respect to the hardware and optimizing the software for increased throughput significantly increases system performance [16].

Note that the goal is to maximize system performance, not MIPS. Traditionally, hardware engineers focus on increasing MIPS, and software engineers focus on reducing path length (instructions per transaction). Most optimizations affect both system performance and MIPS in a positive way—a win–win situation. However, some optimizations increase throughput but decrease MIPS or increase path length. While decreasing MIPS or increasing path length is not very popular with hardware or software engineers, respectively, the focus must remain on the system performance. To achieve this, good teamwork between hardware and software engineers is essential. It is also important for software engineers to perform optimizations that primarily increase MIPS.

An additional benefit gained from having engineers participate in system performance optimization is that the hardware engineers gain knowledge about how software is changing. Because hardware design decisions are made several years before the product ships, and software is continuously evolving, it is important to anticipate changes to the software when making hardware design decisions. The information gained by the hardware engineers can be used to adjust the characteristics of the measurement data to better match the expected characteristics of future software. These adjustments are part of the process described in Section 4. This makes the methodology described in this paper a closed-loop process, as information from system optimization is fed back into the early stages of the design of future hardware (Figure 1).

As mentioned above, cache misses waste a significant portion of the processor execution time. Because

processors are increasing performance faster than memories are decreasing access latency, this will be an increasingly important area for focus over time. Optimizing software to reduce cache misses, especially those from read–write sharing, can significantly increase performance.

Feedback-directed profiling (FDPR) is an optimization that not only improves branch direction, which increases pipeline utilization, but also improves L1 instruction cache-miss rates and instruction misses in the TLB. The information gathered while profiling is used to lay out the code so that the paths most likely to be executed are close together and branches fall through. This means that the instructions most likely to be fetched and executed are in fewer cache lines. The profile information is also used to place together the methods/functions most likely to be executed. This minimizes the number of pages containing "hot" methods/functions and the entries in the TLB needed for the related instructions. This is a good example of how information learned by hardware engineers from software engineers, as a result of working closely together before the optimization was performed, influenced the design of the branching in the processor described in Section 9. For more information on this type of optimization, see the papers by Schmidt et al. [15] and Kalamatianos and Kaeli [32].

Read–write shared data causes cache-to-cache movement of cache lines. This kind of cache miss is problematic because a bigger cache does not reduce its frequency. It exists regardless of cache size. Also, a significant portion of the latency of such misses is determined by wire delay, packaging, and physical distance. Because these factors change little with advancing technology, the latency of these misses consumes a larger number of clock cycles as processor clock frequency increases. These two factors make this an increasingly important component of the CPI affecting the scalability of the multiprocessor MIPS. Hardware traces, as described in Section 3, can be used as input to a multiprocessor cache simulator to find these cache lines and sort them by frequency of movement. These addresses can then be mapped back to their source code classes and objects. This information gives the software engineers very good guidance and prioritization in their efforts to reduce these misses.

Another tool used to attack cache misses, including those caused by read–write sharing, is the PMCs in the processors, which were described in Section 3. These counters can be configured to count cache misses or snoop-hit-modified misses and to cause an overflow interrupt on every *N*th occurrence. Software to capture the instruction and data address associated with the instruction that caused the event can be incorporated in the interrupt handler. This information can produce a profile of the software on the monitored events. This profile also gives the software engineers good guidance when attacking cache misses of all kinds.

Other optimizations performed on the software are aimed at increasing throughput and scalability on multiprocessors. Both of these often come down to optimizing locks protecting shared data structures. The classic lock spins in a loop until the lock becomes available. This spinning increases path length and reduces scalability. Spinning must be kept to a minimum for good scalability. In addition to the classic spinning lock, a number of different types of locks can be used to minimize wasted time [33]. The lock type to choose depends upon the utilization of the lock and how long the lock is held once obtained. In addition to selecting the right lock, there are many ways to make locks more granular. For example, many locks can be created, each protecting a small part of a data structure, rather than one lock protecting the whole data structure. The approach to making locks more granular depends upon the type of data structure(s) being protected and the algorithm that uses the data.

In addition to improving scalability on multiprocessors, algorithms and commonly used paths through the code can be optimized. There are a variety of tools that can provide useful information when performing these optimizations. At the lowest level, it is helpful to have tools that show commonly used paths through the code. At a slightly higher level, a tool that shows the call tree (a sequence of calls from one method to another) with statistics on the frequency of each branch of the tree can be very useful in optimizing algorithms. At an even higher level, tools that show the sequence of task switches and the reason for each switch can also provide useful insight into how the various tasks and algorithms are interacting. Some of the tools and how they are used to optimize the iSeries operating system are described in more detail in a paper by Kunkel et al. [16].

Software optimization extends to other subsystems of commercial servers as well—in particular I/O and memory management. A good example of this relates to disk activity. Overall system throughput increases as the frequency of instructions to handle disk requests decreases. More processor cycles are made available to complete user transactions, and the system becomes more efficient. As with all key software functions, significant effort is made to minimize the path length required to complete each disk request. However, the number of disk accesses required per unit of system work is also optimized. Memory capacity and the software algorithms that manage it play a direct role in this element of performance tuning.

Obviously, the more memory that is available, the more disk accesses can be reduced as page faults decrease.

However, memory can be one of the more costly components for servers, and software algorithms tend to focus on minimizing the required memory capacity. Thus, a tradeoff exists: memory capacity vs. number of disk I/Os vs. path length per disk I/O. Software algorithms manage the optimization of these separate elements so as to meet the performance and price-performance goals of the server design.

11. Summary

In summary, this paper discusses a methodology for analyzing and optimizing the performance of commercial servers. Customer workloads for these systems are shown to have significant variation in their characteristics, creating complexity in designing servers to perform well across the spectrum of applications. The TPC-C workload is highlighted as an example in which many different factors in server performance must be addressed, and tradeoffs evaluated, so as to optimize a design to meet the needs of customers in the marketplace.

The steps in the process of server performance optimization are described and include the following:

1. Selection of representative commercial workloads and identification of key characteristics to be evaluated.
2. Operation of workloads on real systems to collect performance data. Various instrumentation techniques are discussed (software approaches, traces, counters, and hardware emulators), and their limitations are presented.
3. Creation of input data for performance models on the basis of measured workload information. This step in the methodology must overcome the operating environment differences between the instance of the measured system under test and the target system design to be modeled. Various difficulties to consider and techniques to overcome these differences are explained.
4. Creation of performance models. Two general types are described: analytic models and detailed cycle-accurate simulators. Both approaches have their advantages and limitations, which are discussed.
5. Performance prediction and analysis of model results.
6. System performance optimization. Tuning of the operating system and application software is described.

This paper presents a variety of design issues that relate to optimizing the performance of processors, I/O subsystems, and software. From the information provided by models, tradeoffs can be evaluated and optimized, and performance projections can be made with respect to a commercial server for its spectrum of applications.

Throughout this discussion of commercial server performance methodology, a common theme is expressed: Optimization of performance among commercial applications is not simply an exercise in using traces to maximize the processor MIPS. Certainly, processor throughput is an important contributor, but equally significant are items such as using probabilities to reflect future workload characteristics, software tuning for path length, cache miss-rate optimization, memory management, and I/O performance. The paper has presented techniques for evaluating the performance of each of these key contributors so as to optimize the overall performance and cost/performance of commercial servers.

Acknowledgments

We would like to acknowledge Jack Randolph for his years of work on the PMCs and tracing port of the processor chip, David Pease for the development of the interface hardware to collect traces, Men-Chow Chiang for establishing mean value analysis as our preferred technique for high-level modeling, Brad Nelson for his many hours of collecting I/O traces, and Harold Kossman for his many years of leadership of the performance team. We would also like to honor the memory of William Hardell for his contributions to the EZMVA high-level modeling environment.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

References

1. *http://www.tpc.org*.
2. *http://www.notesbench.org*.
3. *http://www.sap-ag.de*.
4. R. Odell and E. Barsness, "IBM AS/400 Business Object Benchmark for Java (jBOB)," IBM White Paper, April 1999, IBM Corporation, available at *http://www.as400.ibm.com/whpapr/jbob400.htm*.
5. M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
6. *The PowerPC Architecture*, Second Edition, C. May, E. Silha, R. Simpson, and H. Warren, Eds., Morgan Kaufmann Publishers, San Francisco, 1994.
7. A. Mink, R. Carpenter, G. Nact, and J. Roberts, "Multiprocessor Performance-Measurement Instrumentation," *IEEE Computer* **23,** 63–75 (1990).
8. A. D. Malony, D. A. Reed, and H. A. G. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Trans. Parallel & Distr. Syst.* **3,** No. 4, 433–450 (1992).
9. C. B. Stunkel, B. Janssens, and W. K. Fuchs, "Address Tracing for Parallel Machines," *IEEE Computer* **24,** 31–38 (1991).
10. J. R. Larus, "Efficient Program Tracing," *IEEE Computer* **26,** 52–61 (1993).
11. A. Agarwal, R. L. Sites, and M. Horwitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th International Symposium on Computer Architecture*, 1986, pp. 119–127.

12. T. R. Puzak, "Cache-Memory Design," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, 1985.
13. L. A. Barroso, S. Iman, M. Dubois, and K. Ramamurthy, "RPM: A Rapid Prototyping Engine for Multiprocessor Systems," *IEEE Computer* **28,** No. 2, 26–34 (1995).
14. A. Nanda, K.-K. Mak, K. Sugavanam, R. K. Sahoo, V. Soundararajan, and T. B. Smith, "MemorIES: A Programmable, Real-Time Hardware Emulation Tool for Multiprocessor Server Design," *Proceedings of the Ninth International Conference on Architecture Support for Programming Languages and Operating Systems*, November 2000, pp. 37–48.
15. W. J. Schmidt, R. R. Roediger, C. S. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky, "Profile-Directed Restructuring of Operating System Code," *IBM Syst. J.* **37,** No. 2, 270–297 (1998).
16. S. Kunkel, B. Armstrong, and P. Vitale, "System Optimization for OLTP Workloads," *IEEE Micro* **19,** 56–64 (1999).
17. K. Keeton, "The Impact of Database System Configuration on Computer Architecture Performance Evaluation," presented in tutorial session with the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.
18. K. M. Chandy, U. Herzog, and L. Woo, "Parametric Analysis of Queuing Networks," *IBM J. Res. Develop.* **19,** 36–42 (1975).
19. S. S. Lavenberg and M. Reiser, "Stationary State Probabilities at Arrival Instants for Closed Queuing Networks with Multiple Types of Customers," *J. Appl. Prob.* **17,** 1048–1061 (1980).
20. M. Chiang and G. S. Sohi, "Experience with Mean Value Analysis Models for Evaluating Shared Bus, Throughput-Oriented Multiprocessors," *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991, pp. 90–100.
21. M. K. Vernon, E. D. Lazowska, and J. Zahorjan, "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols," *Proceedings of the 15th Annual Symposium on Computer Architecture*, Honolulu, 1988, pp. 308–315.
22. M. Reily and J. Edmondson, "Performance Simulation of an Alpha Microprocessor," *IEEE Computer* **31,** No. 5, 50–58 (1998).
23. P. Boos and T. Connate, "Performance Analysis and Its Impact on Design," *IEEE Computer* **31,** No. 5, 41–49 (1998).
24. M. Moudgill, J.-D. Wellman, and J. H. Moreno, "Environment for PowerPC Microarchitecture Exploration," *IEEE Micro* **19,** No. 3, 15–25 (1999).
25. D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz, "Introduction to Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of COMPCON Spring '89*, *Thirty-fourth IEEE Computer Society International Conference*, 1989, pp. 112–117.
26. *http://www.mesquite.com*.
27. G. Ganger, "System-Oriented Evaluation of Storage Subsystem Performance," Ph.D. Dissertation, *Publication No. CSE-TR-243-95*, University of Michigan, Ann Arbor, June 1995.
28. J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel, "A Multithreaded PowerPC Processor for Commercial Servers," *IBM J. Res. Develop.* **44,** No. 6, 885–898 (2000, this issue).
29. A. Maynard, C. Donnelly, and B. Olzewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads," *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1994, pp. 145–156.
30. K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker, "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads," *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, June 1998, pp. 15–26.
31. L. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, June 1998, pp. 3–14.
32. J. Kalamatianos and D. Kaeli, "Temporal-Based Procedure Reordering for Improved Instruction Cache Performance," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, January 1998, pp. 244–253.
33. P. Magnusson, A. Landin, and E. Hagersten, "Queue Locks on Cache Coherent Multiprocessors," *Proceedings of the Eighth International Parallel Processing Symposium*, April 1994, pp. 165–171.

Received February 2, 2000; accepted for publication November 20, 2000

Steven R. Kunkel *IBM Server Group, 3605 Highway 52 N, Rochester, Minnesota 55901 (srkunkel@us.ibm.com).* Dr. Kunkel received his Ph.D. degree from the University of Wisconsin at Madison in 1987. He then joined IBM in Endicott, New York, doing performance analysis of a vector facility for a mid-range S/390 product. In 1989, he transferred to Rochester, Minnesota, where he has worked on architecture and performance analysis for AS/400 products, including such areas as NUMA, VLIW, caches, MP cache coherency, SCI, multithreading, and converting AS/400 to PowerPC-architecture processors. Dr. Kunkel is currently a Senior Technical Staff Member doing architecture and performance analysis for iSeries (AS/400), pSeries (RS/6000), and xSeries (Netfinity) servers.

Richard J. Eickemeyer *IBM Server Group, 3605 Highway 52 N, Rochester, Minnesota 55901 (eick@us.ibm.com).* Dr. Eickemeyer is a Senior Engineer in the IBM Server Group. He is currently the processor core performance team lead for IBM PowerPC servers. Prior to this, he worked on the performance and architecture of several processors used in AS/400 systems and S/390 systems in Rochester, Minnesota, and Endicott, New York. Since joining IBM, he has received awards which include the Seventh Plateau IBM Invention Achievement Award, an Outstanding Technical Achievement Award, an Outstanding Innovation Award, and a Corporate Award for Hardware Multi-Threading. He has also been named a Server Group Master Inventor. Dr. Eickemeyer received the B.S. degree in electrical engineering from Purdue University and the M.S. and Ph.D. degrees from the University of Illinois at Urbana–Champaign. His research interests are computer architecture and performance analysis.

Mikko H. Lipasti *University of Wisconsin at Madison, 4613 Engineering Hall, 1415 Engineering Drive, Madison, Wisconsin 53706 (mikko@engr.wisc.edu).* Dr. Lipasti has worked for IBM in both software and future processor and system performance analysis and design guidance, as well as operating system kernel implementation. He has published several conference and journal papers, primarily in the area of value prediction, filed seven patent applications, won the Best Paper Award at MICRO-28, and received IBM Invention Achievement, Patent Issuance, and Technical Recognition awards. His research interests include operating systems, compiler optimization, commercial workloads, and the interaction of these with computer system architecture and microarchitecture. Dr. Lipasti joined the Department of Electrical and Computer Engineering at the University of Wisconsin at Madison as an Assistant Professor in August 1999.

Timothy J. Mullins *IBM Server Group, 3605 Highway 52 N, Rochester, Minnesota 55901 (mullinst@us.ibm.com).* Mr. Mullins joined the Rochester Development Laboratory after receiving his B.S.E.E. degree from the University of California at Berkeley in 1977. In 1982, he received his M.S.E.E. degree from the University of Minnesota. He has done work in I/O controller design and in CPU development in the areas of logic design and timing analysis. Since 1986, he has been involved in various assignments relating to computer system performance analysis, including processors, I/O design, system buses, storage subsystems, and system designs. Mr. Mullins is currently a Senior Technical Staff Member in the Rochester Laboratory's Future Processor Performance Department; he is involved in system design and performance analysis for Server Group products.

Brian O'Krafka *Sun Microsystems, MS AUS08, 5300 Riata Park Court, Austin, Texas 78727 (okrafka@central.sun.com).* Dr. O'Krafka joined the IBM Austin Laboratory in 1992 after receiving his Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley. From 1992 to 1997, he worked on multiprocessor verification for RS/6000 servers. In 1997 he joined the RS/6000 performance group, where he worked on multiprocessor performance modeling and analysis. In 2000 Dr. O'Krafka joined Sun Microsystems, where he is now doing performance analysis of future Sun commercial servers.

Harold Rosenberg *Sun Microsystems, One Network Drive, MS UBUR03-212, Burlington, Massachusetts 01803 (haroldr.Rosenberg@east.sun.com).* Mr. Rosenberg worked as an Advisory Engineer in the IBM Server Group. In that position his main focus was on I/O performance, including storage subsystems and I/O interconnects. He holds a B.S. degree in electrical engineering from Tufts University and an M.S. degree in electrical and computer engineering from the University of Massachusetts; and he has performed additional graduate work, through Ph.D. candidacy in computer science, at the University of Michigan, where his research interests included fault-tolerant computing and dependability evaluation. He previously worked at Digital Equipment Corporation as an ASIC designer in the Storage Subsystems Group. Mr. Rosenberg currently works at Sun Microsystems as an I/O Performance Architect.

Steven P. VanderWiel *IBM Server Group, 3605 Highway 52 N, Rochester, Minnesota 55901 (svw@us.ibm.com).* Dr. VanderWiel is a member of the Future Processor Performance Department of the IBM Server Group, where he analyzes design alternatives for next-generation server systems including the iSeries and pSeries eServers. He received a B.S. degree and an M.S. degree, both in computer engineering, from Iowa State University, and a Ph.D. degree in electrical engineering from the University of Minnesota.

Philip L. Vitale *IBM Server Group, 3605 Highway 52 N, Rochester, Minnesota 55901 (vit@us.ibm.com).* Mr. Vitale is an Advisory Engineer with the IBM Server Group. He specializes in performance instrumentation, workload analysis, and the development of technology to unite hardware and software optimization efforts. He holds a master's degree in computer science from the University of Wisconsin at Madison.

Larry D. Whitley *IBM Server Group, 3605 Highway 52 N, Rochester, Minnesota 55901 (ldw@us.ibm.com).* Mr. Whitley is a Senior Engineer in the RS/AS Hardware Performance group at the Rochester, Minnesota, facility of the IBM Server Group. After receiving a B.S. degree in electrical engineering from the University of Missouri at Columbia in 1969, he joined IBM and has worked on the design of processors, I/O, and system control programming for several IBM systems, including the System/32, System/34, and System/36. More recently, as a part of the design process, he has created performance models of processor memory subsystems, I/O subsystems, and I/O interconnection fabrics for the AS/400, RS/6000, and Netfinity servers. Mr. Whitley has received Outstanding Technical Contribution awards and holds three patents.