Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads

Cambridge, Massachusetts

February 2, 2002

Immediately precedes the

Eighth International Symposium on High Performance Computer Architecture

Sponsored by the IEEE Computer Society

Organized by:

Russell Clapp, IBM

rclapp@us.ibm.com

Kimberly Keeton, Hewlett-Packard Laboratories

kkeeton@hpl.hp.com

Ashwini Nanda, IBM TJ Watson Research Center

ashwini@watson.ibm.com

Final Program

8:00 am - 8:15 am

Registration

8:20 am - 8:30 am

Introductory Comments

8:30 am – 10:00 am

Session 1: Invited Talks and Benchmarking

Evaluation of Shared Cache Architectures for TPC-H

Michel Dubois, Jaeheon Jeong, Shahin Razeghia, Mahsa Rouhaniz and Ashwini Nanda
University of Southern California and IBM

Precise and Accurate Processor Simulation

Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz and Mikko H. Lipasti
University of Wisconsin - Madison

New Challenges in Benchmarking Future Processors

Shubhendu S. Mukherjee
Intel Corporation

10:00 am - 10:30 am

Coffee Break

10:30 am – 12:00 pm

Session 2: Methodologies

Evaluating Non-deterministic Multi-threaded Commercial Workloads

Alaa R. Alameldeen, Pacia J. Harper, Milo M. K. Martin, Carl J. Mauer, Daniel J. Sorin,
Min Xu, Mark D. Hill and David A. Wood
University of Wisconsin - Madison

How Input Data Sets Change Program Behaviour

Lieven Eeckhout, Hans Vandierendonck, Koen De Bosschere
Department of Electronics and Information Systems (ELIS)
Ghent University – Belgium

Benchmarking Web Server Architectures: A Simulation Study on Micro Performance

Haiyong Xie, Laxmi Bhuyan and Yeim-Kuan Chang
Department of Computer Science & Engineering
University of California - Riverside

12:00 am - 1:30 pm

Lunch

1:30 pm – 3:00 pm

Session 3: Architecture Evaluation and Modeling

Compressibility Characteristics of Address/Data Transfers in Commercial Workloads

Krishna Kant and Ravi Iyer
Enterprise Architecture Laboratory
Intel Corporation

Performance Workloads in a Hardware Multi Threading Environment

Bret Olszewski and Octavian F. Herescu
IBM

A Processor Queuing Simulation Model for Multiprocessor System Performance Analysis

Thin-Fong Tsuei and Wayne Yamamoto
Sun Microsystems

3:00 pm - 3:30 pm

Coffee Break

3:30 pm - 5:00 pm

Session 4: Workload Characterization

Performance Analysis of Speech Recognition Software

Chunrong Lai, Shih-Lien Lu and Qingwei Zhao
Intel Corporation

Comparison of Memory System Behavior in Java and Non-Java Commercial Workloads

Morris Marden, Shih-Lien Lu, Konrad Lai Mikko Lipasti
University of Wisconsin – Madison and Intel Corporation

Characterizing TPC-H on a Clustered Database Engine from the OS Perspective

Yanyong Zhang, Jianyong Zhang, Anand Sivasubramaniam, Chun Liu and Hubertus Franke
The Pennsylvania State University and IBM T.J. Watson Research Center

5:00 pm

Participant Feedback

Closing Remarks

Session 1

Invited Talks and Benchmarking

Evaluation of Shared Cache Architectures for TPC-H

Michel Dubois, Jaeheon Jeong, Shahin Razeghia,
Mahsa Rouhaniz and Ashwini Nanda
University of Southern California and IBM

Precise and Accurate Processor Simulation

Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz
and Mikko H. Lipasti
University of Wisconsin – Madison

New Challenges in Benchmarking Future Processors

Shubhendu S. Mukherjee
Intel Corporation

Evaluation of Shared Cache Architectures for TPC-H

Michel Dubois, Jaeheon Jeong**, Shahin Jarhomi, Mahsa Rouhanizadeh and Ashwini Nanda*

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA90089-2562
dubois@paris.usc.edu

*IBM T.J. Watson Research Center
Yorktown Heights, NY

**IBM
Beaverton, OR97006

1. Introduction

The design of large-scale servers must be optimized for commercial workloads and web-based applications. These servers are high-end, shared-memory multiprocessor systems with large memory hierarchies, whose performance is very workload dependent. Realistic commercial workloads are hard to model because of their complexity and their size.

Trace-driven simulation is a common approach to evaluate memory systems. Unfortunately, storing and uploading full traces for full-size commercial workloads is practically impossible because of the sheer size of the trace. To address this problem, several techniques for sampling traces and for utilizing trace samples have been proposed [2][4][5][7][9].

For this study, we have collected time samples obtained from an actual multiprocessor machine running TPC-H [8] using the MemorIES board developed by IBM [6]. MemorIES was originally designed to emulate memory hierarchies in real-time and is plugged into the system bus of an IBM RS/6000 S7A SMP system running DB2 under AIX. This system can run database workloads with up to 1TB of database data.

The target system that we evaluate with the TPC-H samples is shown in Figure 1. It is an 8-processor bus based machine. Each processor has 8MB of second level cache (4-way, 128 byte blocks). Cache coherence is maintained by snooping on a high-speed bus. The 24GByte main memory is fronted by a shared cache [11], whose goal is to cut down on the effective latency of the large and slow DRAM memory. One advantage of a shared cache is that it does not require cache coherence. The disadvantages are that L2-miss penalties are only partially reduced and the shared cache does not bring any relieve to bus traffic.

We look at the effects of cache block size and cache organization. We observe that IO has a large impact on the cache behavior, especially for very large shared caches (e.g., 1 GBytes) and therefore we evaluate various strategies for handling IO bus request in the shared cache --invalidate, update and allocate.

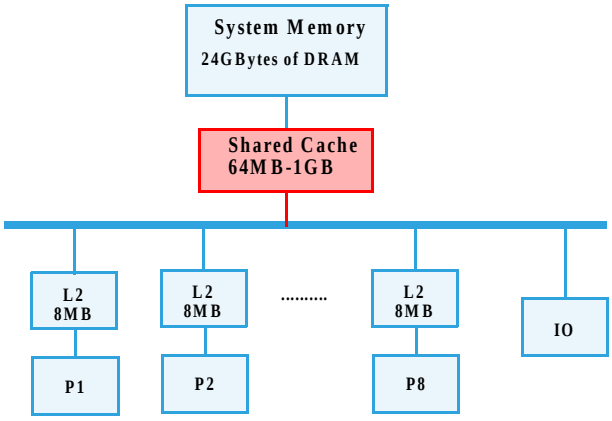


Figure 1. Target Multiprocessor System with Shared Cache

The rest of this paper is organized as follows. In Section 2, we present the tracing tool obtained by configuring MemorIES, the trace sampling scheme, and the characteristics of the trace samples. In Section 3, we describe the target shared-cache architectures. In Section 4, 5 and 6, we show the impact of the cold-start effect and classify miss rates into sure maximum and unknown miss rates. Based on that classification, we are able to draw some conclusions on the effectiveness of the shared cache for TPC-H. Finally we conclude this abstract in Section 7.

2. TPC-H Traces

Much of this section is reproduced from [6] and [3] and is included here for completeness.

2.1. Tracing Environment

The IBM Memory Instrumentation and Emulation System (*MemorIES*) was designed to evaluate trade-

offs for future memory system designs in multiprocessor servers. The MemorIES board sits on the bus of a conventional SMP and passively monitors all the bus transactions, as shown in Figure 2. The host machine is an IBM RS/6000 S7A SMP server, a bus-based shared-memory multiprocessor. The server configuration consists of eight Northstar processors running at 262MHz and a number of IO processors connected to the 6xx bus operating at 88 MHz. Each processor has a private 8 MB 4-way associative L2 cache. The cache block size is 128 bytes. Currently the size of main memory is 24 Gbytes.

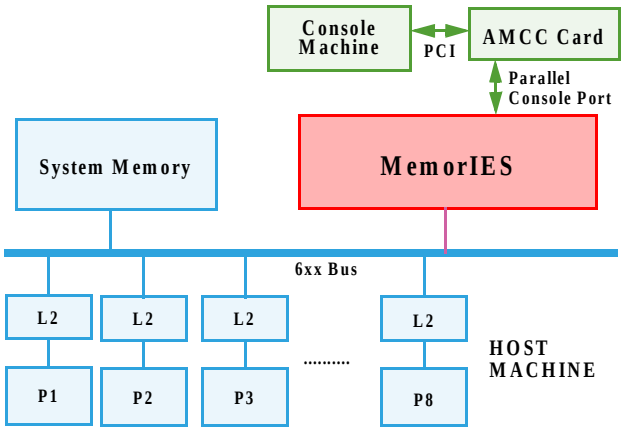


Figure 2. Operating environment of MemorIES

A finely tuned 100Gbyte TPC-H benchmark [8] runs on top of DB2 under AIX. Its execution takes three days on the server.

Although it has been conceived for online cache emulation, MemorIES can be configured as a tracing tool. In this mode, it captures bus transactions in real time and stores them into its on-board memory. Later on, as the on-board memory fills up, the trace is uploaded to a disk. Currently, with its 1GB of SDRAM, MemorIES can collect up to 128M bus transaction records without any interruption, since each record occupies 8 bytes.

2.2. Trace Samples

Although MemorIES can store up to 128M references per sample, the size of each sample collected for this study is 64M references. Currently, it takes about one hour to dump to disk a sample of 64M references or 512Mbytes. Our trace consists of 12 samples with roughly one hour between samples and its overall size is 6 Gbytes. The trace samples were taken during the first day of a three-day execution of a 100GB TPC-H. Each sample records about a few minutes of execution.

Table 1 shows the reference counts of every processor and for every transaction type in the first trace sample. It shows that the variance of the reference count among processors (excluding IO processors) is very small. The references by IO processors are dominated by write/flush. We found similar distributions in the rest of the samples.

Table 2 shows the reference counts of eight transaction types in each of the 12 trace samples. We focus on the following classes of bus transactions: reads, writes (read-excl, write/inv), upgrades, and write/flushes (essentially due to IO write). In the 12 samples, processor reads and writes (including upgrades) contribute to around 44% and 21% of all bus transactions, respectively. 35% of all bus transactions are due to IO-writes. IO-writes are caused by input to memory. On each IO-Write, the data block is transmitted on the bus and the processor write-back caches are invalidated to maintain coherence with IO.

We also found some noticeable variations in some reference counts across trace samples. For instance, notice the wide variation in the number of upgrades among samples.

3. Target Shared Cache Architectures

We focus on the evaluation of L3 cache architectures shared by an 8-processor SMP. The shared cache size varies from 64MB to 1GB. Throughout this paper, LRU replacement policy is used in the shared cache. We evaluate Direct mapped and 4-way cache organizations. We also look at two block sizes, 128 Bytes and 1 Kbytes.

The idea of a shared cache is not new as it was proposed and evaluated in [11], in a different environment and using analytical models. The goal of a shared cache as shown in Figure 1 is to cut down on the latency of DRAM accesses to main memory. On every bus cycle, the shared cache is looked up and the DRAM memory is accessed in parallel. If the shared cache hits, then the memory access is aborted. Otherwise, the memory is accessed and the shared cache is updated. Let H be the cache hit rate and let L_{cache} and L_{DRAM} be the latency of a shared cache hit and of a DRAM access. Then the apparent latency of the large DRAM memory is: $(1- H) \times L_{DRAM} + H \times L_{cache}$.

The shared cache can have a high hit ratio even if its size is less than the aggregate L2 cache size in the

PID	read	read-excl	clean	ikill	write/inv	write/clean	upgrade	write/flush
0	3944601	221999	0	192	295218	5677	1629223	67102
1	4001991	217725	64	160	270918	5096	1685160	64724
2	4175661	208862	0	96	303913	5267	1742245	65406
3	3908973	213590	0	0	286916	5430	1610544	66424
4	4101762	209785	64	128	254847	5080	1640377	66063
5	3932938	217326	0	64	314963	5100	1580184	65686
6	4166149	192101	0	0	248305	4533	1821616	65143
7	3851738	200694	0	288	225611	4758	1654915	64449
IO	978887	0	0	0	12	0	0	16232121

TABLE 1. Breakdown of references in the first trace sample in each processor

Sample	read	read-excl	clean	ikill	write/inv	write/clean	upgrade	write/flush
1	33062700	1682082	128	928	2200703	40941	13364264	16757118
2	29574629	1446244	0	2272	2677942	43781	10059975	23304021
3	28717853	1335718	0	0	2835185	35304	10637496	23547308
4	27727985	1405457	6432	6720	2900043	54407	8734182	26273638
5	25799000	1774029	0	0	3067320	51339	9517591	26899585
6	28760710	4021657	96	96	6077446	44147	6367359	21837353
7	29087095	1959379	0	608	2953071	57934	7265500	25785277
8	28830726	1929086	0	160	2866096	59072	7312608	26111116
9	26092006	1970931	96	320	3206843	57642	7186243	28594783
10	26899985	1768684	72	66	3279821	50629	7148337	27961270
11	36070868	1079535	0	0	2009423	96048	10462970	17390020
12	36516311	1108966	50560	51936	2159616	145164	10631200	16445111
Avg (%)	**44.35**	**2.66**	**0.01**	**0.01**	**4.50**	**0.09**	**13.50**	**34.88**

TABLE 2. Reference count per transaction type in the 12 trace samples

multiprocessor, because the shared cache does not maintain inclusion and also because of sharing.

If the main (DRAM) memory is banked and interleaved then a separate shared cache can be used in conjunction with each memory bank. This partitioning of the shared cache may result in lower hit rates than the ones reported here. We assume throughout that the shared cache can buffer any main memory block.

IO transactions can have a huge impact on shared cache performance. The standard way to deal with IO transactions is to simply *invalidate* the cache on each IO transaction detected on the bus. However, other policies are possible, since, on input the transaction carries the data with it. The first policy is IO-*update*, whereby if the shared cache hits, then the block is updated instead of invalidated. The second policy is IO-*allocate*, whereby misses due to IO cache accesses allocate a block in the cache. Because allocating means taking a miss in the shared cache, the control for the cache may be more complex, especially when the bus transaction does not carry the values for the entire block. These two strategies may pay off if the shared cache is big enough to retain database record between the time it is inputted and the time it is accessed.

In the shared cache we simulate, we consider each bus transaction as follows:

`Processor requests`:
`Read`: In case of a hit in the shared cache, we update the LRU statistics for the block. On a miss the data is loaded in cache from memory.
`Read-excl`: same as read.
`Clean,I-kill`: Invalidate the block in the shared cache.
`W-Clean`: same as read.
`W-kill,Dclaim`: These are upgrade request. Treated as reads in the shared cache.

`Write-Flush`: Invalidate the block.

`IO requests:`

There are 3 types of IO requests: IO Read, IO Write and IO Write Flush. By far write flushes are the dominant request type. The request are treated differently, depending to the policy to deal with IO requests.

In IO-invalidate, the shared cache is invalidated on all IO references. In IO-update, the block is updated in the case of a hit and its LRU statistics are updated. In IO-allocate the IO references are treated as if they were coming from execution processors.

The miss rate counts reported in this abstract do not include misses due to IO since they are not on the critical path of processors' execution.

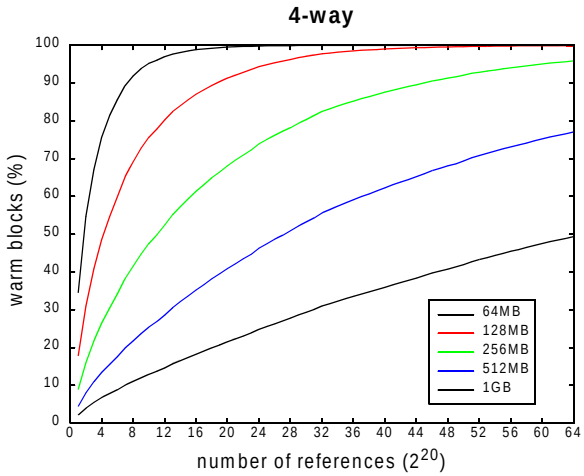


Figure 3. Blockframe Warm-up

4. Cache Warm-up Rates

4.1. Cache Blockframe Warm-up Rates

Figure 3 shows the average fraction of warm blockframes as a function of the reference number across all 12 samples for each of the four clustering schemes. We simulate each trace sample starting with empty caches. A blockframe is deemed warm after a block has been allocated to it for the first time. For every number of references, we calculate the average number of warm blockframes across all samples. The shared cache size varies from 64MB to 1GB. From the graphs, we see that shared caches larger than 128MB are never completely warm with 64M references.

4.2. Cache Set Warm-up Rate

Figure 4 shows the warm-up rate of cache sets. A cache set is deemed warm once all its blockframes have been accessed at least once. The graphs indicate that the cache set warm-up rate is much lower than the cache blockframe warm-up rate for large caches. This implies that cold blockframes are spread among many cache sets.

If the trace-driven simulation of very large shared caches only uses the references to warm cache sets, the evaluation will be naturally biased to a (relatively) small number of hot cache sets. Thus this approach is debatable since a good set sampling strategy might produce a more usable trace. Thus collecting statistics on warm sets only as was proposed in [5] is not acceptable.

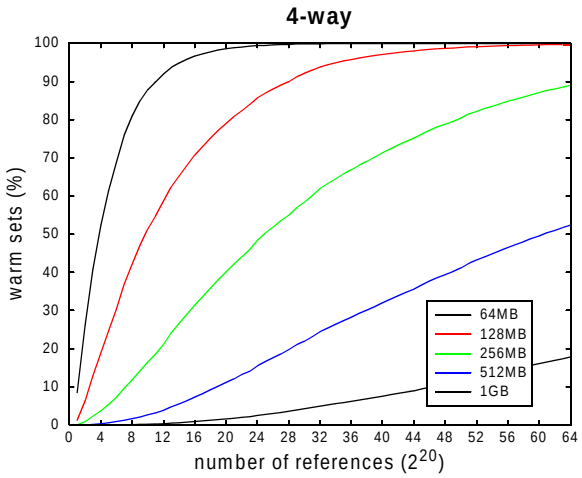


Figure 4. Set Warm-up Rate

5. Shared Cache Miss Rate Classification

To evaluate the performance of shared cache architectures with the trace samples, we use the miss rate as a metric. We decompose misses into the following four categories: cold misses, coherence misses due to IO, replacement misses, and upgrade misses. To do this, we first check whether a block missing in the cache was previously referenced. If it was not and is not preceded by an IO invalidation, the miss is classified as a cold miss. If the missing block was the target of an IO-invalidation before the miss is a deemed a coherence miss due to IO, irrespective of other reasons that may have caused the miss. Otherwise, the miss is due to replacement.

An upgrade miss happens when an upgrade request

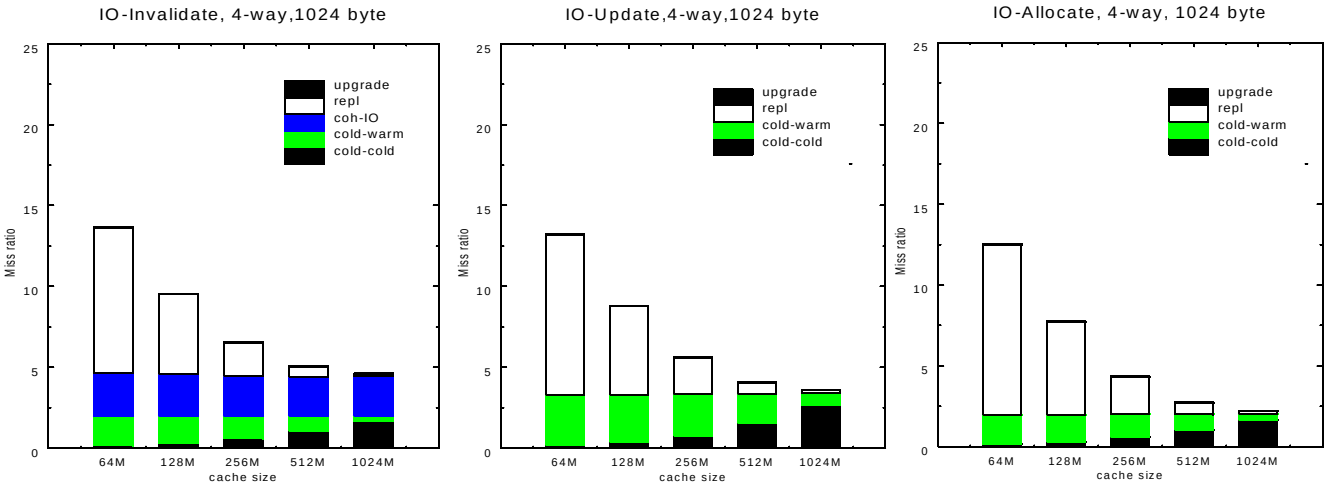


Figure 5. Miss rate classification

on the bus misses in the shared cache. Upgrade misses do not occur when cache inclusion between the processors' L2 caches and the shared cache is maintained, which is not the case here.

To understand the cold-start bias better, cold misses are further decomposed into cold-cold misses and warm-cold misses. A cold-cold miss occurs on a cold miss such that a cache blockframe is allocated in a set that is not filled yet. A warm-cold miss occurs on a cold miss in a set that is already filled.

The reason for separating the types of cold misses is that cold-cold misses may not be actual misses if the trace was continuous. These misses are called *unknown* references in [10]. On the other hand warm-cold misses are *sure* misses, although we don't know what kind of miss they really would be in the full trace.

We analyze the components of the miss rate assuming that the shared cache is empty at the beginning of each sample. To calculate the miss rate, we apply each trace sample to empty caches and then we sum up all the misses and legitimate memory references (the ones listed in Table 1) in each sample.

Figure 5 for 1KByte blocks shows that a large fraction of misses are due to IO coherence in the IO-invalidate system. This component is not present in IO-update or IO-allocate.

As the cache size increases, cold-cold misses become a dominant part of the cold miss component, and the large number of cold-cold misses in very large caches shows that the error due to cold start is significant for these systems.

6. Sure vs. Unknown References

An interesting classification is between sure and unknown references [10]. Sure references are references such that we know for sure that they would hit or miss in the full trace. Unknown references are references for which we cannot decide whether they would hit or miss in the full trace. Of course all hits are sure references. Cold-cold misses that are not preceded by an IO-invalidate are unknown references. All other misses are sure references.

Thus we can classify the miss rates as sure miss rate or unknown miss rate. The maximum miss rate is the ratio of the number of all misses and the number of all references. The sure miss rate is the lower bound on the miss rate. It is the ratio of the number of sure misses by the number of sure references. The unknown miss rate is the difference between the maximum miss rate and the sure miss rate. In the IO-invalidate case, the sure miss rate can be decomposed in the sure miss rate due to IO and other sure miss rate. This is shown in Figure 6. The white component at the top is the zone of uncertainty. That's what we also call the unknown miss rate. It shows the error range for the results based on our samples.

The sure miss rate in Figure 6 (in blue and red) is a lower bound on the expected miss rate. The assumption for this lower bound is that the unknown references have the same hit/miss behavior in the full trace as other references. However, it has been shown that the unknown references tend to miss more than others [10]. The black line superimposed to the stack bar

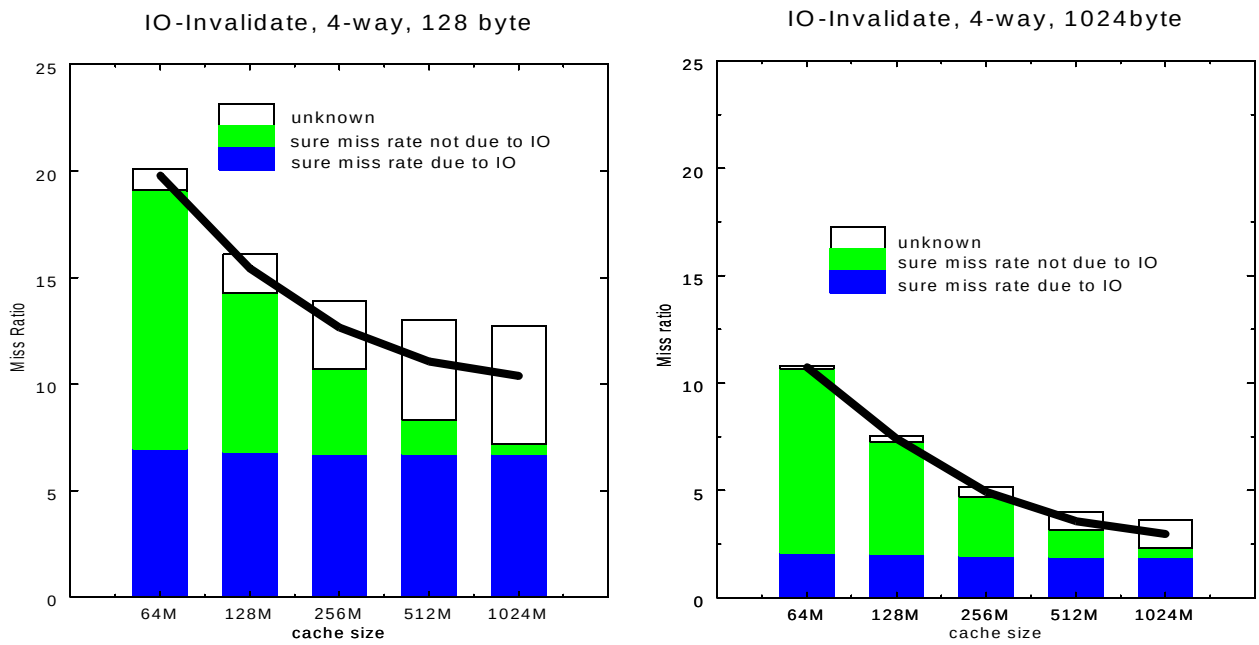


Figure 6. Miss Rates of a 4-way Shared Cache with IO Invalidate

chart shows the average between the maximum miss rate and the sure miss rate. In [10] it was shown to be a pretty good estimate of the actual miss rate.

When the block size is 128B, the zone of uncertainty is quite large, but for 1KB blocks, the zone of uncertainty is small. Obviously, under IO-invalidate the 1KB block size is a much better choice and it definitely pays to add more shared cache in this case.

Figure 7 shows the impact of the strategy for IO bus requests. When the block size is 128 bytes, the strategy used for IO bus requests does not make much difference and this is a reliable result, since the zone of uncertainty is very small. However, when the block size is 1Kbytes, we see that potentially IO-update and IO-allocate strategies could make a big difference, but the zone of uncertainty is too large in this case to be conclusive.

Finally Figure 8 shows the effect of the block size and the cache organization. We see that the block size has a huge impact on performance but that the organization of the cache does not matter much. Obviously conflict misses are few and far between. This is a reliable conclusion given the small size of the zones of uncertainty.

7. Conclusion

In this work, we have used 12 time samples of TPC-H collected during its first day of execution. The trace samples have been used to evaluate various architectures for a shared caches in a typical SMP system with 8 processors. We have looked at block sizes, cache sizes, cache organizations and strategies to handle IO requests. Because of cold start effects in each sample, we are not able to draw definite conclusions on all aspects of the shared cache architecture. Cold start effects in each sample result in a zone of uncertainty due to unknown references.

We have observed the following:

- Even small shared caches of 64 Mbytes (equal to the aggregate L2 cache size) is very effective, due to sharing and non inclusion
- A block size of 1Kbyte is much better than a block size of 128 bytes.
- The cache organization does not affect performance much. Thus a simple Direct mapped cache is probably preferable. This indicates few conflicts in the shared cache
- With a 1GB cache and a block size of 1KB the shared cache miss rate is only a few percents
- It would appear that for larger caches the strategy for handling IO requests (IO-invali-

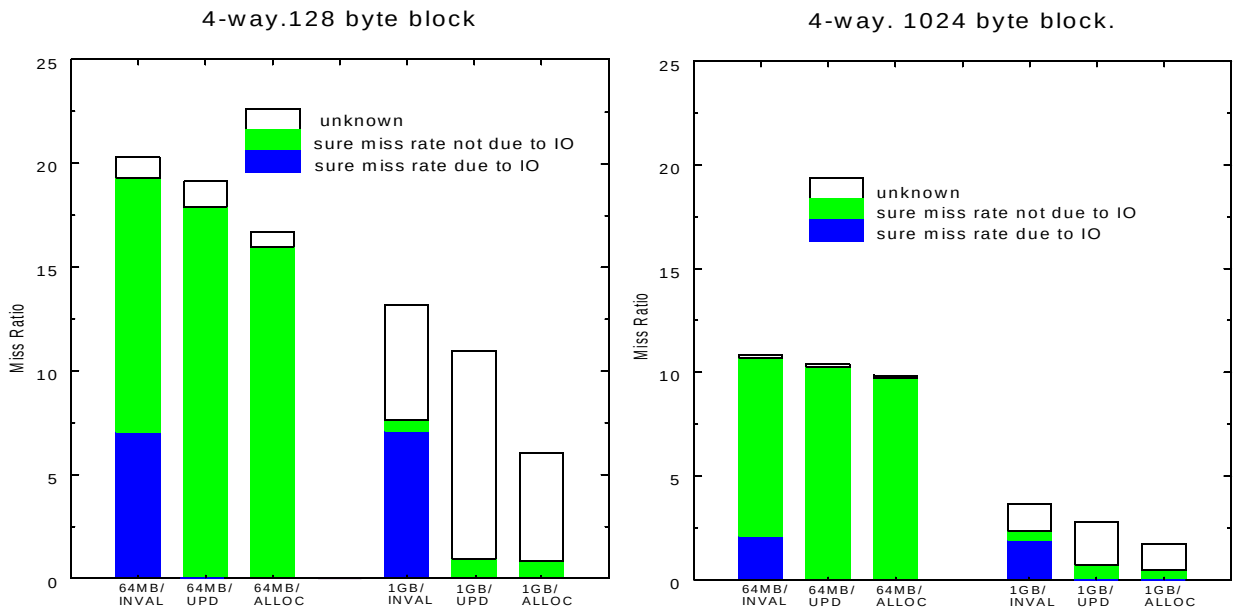


Figure 7. Miss Rates Comparison Between Various Cache Strategies for IO Requests.

date, IO-update or IO-allocate) may have a large impact on the miss rate. However the zone of uncertainty is too large to draw definite conclusions.

We are trying to narrow the zone of uncertainty to get more conclusive evidence. Since the MemorIES board can emulate target cache systems in real time, we can use the time between samples to emulate a set of caches with different architectures and fill these caches up before the next sample is taken so that we have the content of these caches at the beginning of each sample. The content of these emulated caches is dumped with the sample at the end of each time sample. By playing with cache inclusion properties [9], the content of these few emulated caches can be used to restore the state of many different cache configurations at the beginning of each sample, thus eliminating the cold start effect. In this framework, a trace collection experiment consists of several phases, repeated for each time sample: a phase in which we emulate the target caches to get the snapshots, the trace sample collection phase, and the trace dump phase in which the snapshots and the trace samples are dumped to disk.

Once we have the trace with the cache snapshots we will be able to firm up the conclusions of this paper.

8. Acknowledgments

This work is supported by NSF Grant CCR-0105761 and by an IBM Faculty Partnership Award. We are grateful to Ramendra Sahoo and Krishnan Sugavanam from IBM Yorktown Heights who helped us obtain the trace.

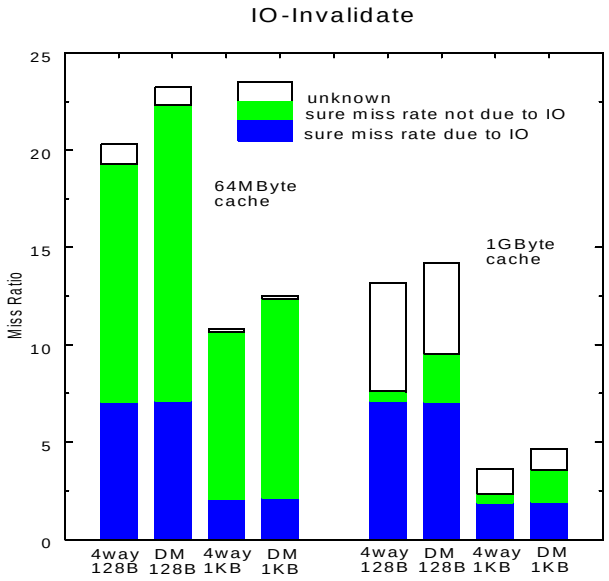


Figure 8. Effect of Cache Organization and Block Size

9. References

[1] L. Barroso, K. Gharachorloo and E. Bugnion, "Memory System Characterization of Commercial Workloads," In *Proceedings of the 25th ACM International Symposium on Computer Architecture*, June 1998.

[2] J. Chame and M. Dubois, "Cache Inclusion and Processor Sampling in Multiprocessor Simulations," In *Proceeding of ACM Sigmetrics*, pp. 36-47, May 1993.

[3] J. Jeong, R. Sahoo, K. Sugavanam, A. Nanda and M. Dubois, "Evaluation of TPC-H Bus Trace Samples Obtained with MemorIES" *Workshop on Memory Performance Issues, ISCA 2001, http:// www.ece.neu.edu/conf/wmpi2001/full.htm*.

[4] R. Kessler, M. Hill and D. Wood, "A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches," *IEEE Transactions on Computers*, vol. 43, no. 6, pp. 664-675, June 1994.

[5] S. Laha, J. Patel, and R.K. Iyer, "Accurate Low-cost Methods for Performance Evaluation of Cache Memory Systems, *IEEE Transactions on Computers*, pp. 1325-1336, Vol. 37, No. 11, Nov. 1988.

[6] A. Nanda, K. Mak, K. Sugavanam, R. Sahoo, V. Soundararajan, and T. Basil Smith, "MemorIES: A Programmable, Real-Time Hardware Emulation Tool for Multiprocessor Server Design," In *Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[7] T. R. Puzak, "Analysis of Cache Replacement-Algorithms," *Ph.D. Dissertation*, Univ. of Massachusetts, Amherst, MA., Feb. 1985.

[8] Transaction Processing Performance Council, "TPC Benchmark H Standard Specification," *Transaction Processing Performance Council*, June 1999. http://tpc.org.

[9] W. Wang and J. L. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," *ACM Transactions on Computer Systems*, vol. 9, no. 3, pp. 222-241, August 1991.

[10] D. Wood, M. Hill, and R. Kessler, "A Model for Estimating Trace-Sample Miss Ratios," In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 27-36, 1990.

[11] P. Yeh, J. Patel, and Ed. Davidson, "Performance of Shared Cache for Parallel-Pipelined Computer Systems," In *Proceedings of the ACM International Symposium on Computer Architecture*, pp. 117-123, 1983.

Precise and Accurate Processor Simulation

Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko H. Lipasti

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI 53706
{cain,baschwar}@cs.wisc.edu

Electrical and Computer Engineering
University of Wisconsin
1415 Engineering Drive
Madison, WI 53706
{lepak,mikko}@ece.wisc.edu

Abstract

Precise and accurate simulation of processors and computer systems is a painstaking, time-consuming, and error-prone task. Abstraction and simplification are powerful tools for reducing the cost and complexity of simulation, but are known to reduce precision. Similarly, limiting and simplifying the workloads that are used to drive simulation can simplify the task of the computer architect, while placing the accuracy of the simulation at risk. Historically, precision has been favored over accuracy, resulting in simulators that are able to analyze minute machine model variations while mispredicting—sometimes dramatically—the actual performance of the processor being simulated. In this paper, we argue that both precise and accurate simulators are needed, and provide evidence that counters conventional wisdom for three factors that affect precision and accuracy in simulation. First, we show that operating system effects are important not just for commercial workloads, but also for SPEC integer benchmarks. Next, we show that simulating incorrect speculative paths is largely unimportant for both commercial workloads and SPEC integer benchmarks. Finally, we argue that correct simulation of I/O behavior, even in uniprocessors, can affect simulator accuracy.

1.0 Introduction and Motivation

For many years, simulation at various levels of abstraction has played a key role in the design of computer systems. There are numerous compelling reasons for implementing simulators, most of them obvious. Design teams need simulators throughout all phases of the design cycle. Initially, during high-level design, simulation is used to narrow the design space and establish credible and feasible alternatives that are likely to meet competitive performance objectives. Later, during microarchitectural definition, a simulator helps guide engineering trade-offs by enabling quantitative comparison of various alternatives. During design implementation, simulators are employed for testing, functional validation, and late-cycle design trade-offs. Finally, simulators provide a useful reference for performance validation once real hardware becomes available.

Outside of the industrial design cycle, simulators are also heavily used in the computer architecture academic research community. Within this context, simulators are primarily used as a vehicle for demonstrating or comparing the utility of new architectural features, compilation techniques, or microarchitectural techniques, rather than for helping to guide an actual design project. As a result, academic simulators are rarely used for functional or performance validation, but strictly for proof of concept, design space exploration, or quantitative trade-off analysis.

Simulation can occur at various levels of abstraction. Some possible approaches and their benefits and drawbacks are summarized in Table 1. For example, flexible and powerful analytical models that exploit queueing theory can be used to examine system-level trade-offs, identify performance bottlenecks, and make coarse performance projections. Alternatively, equations that compute cycles per instruction (CPI) based on cache miss rates and fixed latencies can also be used to estimate performance, though this approach is not effective for systems that are able to mask latency with concurrent activity. These approaches are powerful and widely employed, but will not be considered further in this paper. Instead, we will focus on the last three alternatives: the use of either trace-driven, execution-driven, or full-system simulation to simulate processors and computer systems.

Trace-driven simulation utilizes execution traces collected from real systems. Collection schemes range from software-only schemes that instrument program binaries all the way to proprietary hardware devices that connect to processor debug ports. The former pollute the collected trace, since the software overhead slows program execution relative to I/O devices and other external events. The latter can run at full speed, but require expensive investment in proprietary hardware, knowledge of debug port interfaces, and are probably not feasible for future multi-gigahertz processors. Trace collection is also hampered by the fact that usually only non-speculative or committed instructions are recorded in the trace. Hence, the effects of speculatively executed instructions from incorrect branch paths are lost. Furthermore, once a trace has been col-

Modeling Technique	Inputs	Benefits	Drawbacks
Analytical models	Cache miss rates; I/O rates	Flexible, fast, convenient, provide intuition	Cannot model concurrency; lack of precision
CPI Equations	Core CPI, cache miss rates	Simple, intuitive, reasonably accurate	Cannot model concurrency; lack of precision
Trace-driven Simulation	Hardware traces; software traces	Detailed, precise	Trace collection challenges; lack of speculative effects; implementation complexity
Execution-driven Simulation	Programs, input sets	Detailed, precise, speculative paths	Implementation complexity; simulation time overhead; correctness requirement; lack of OS and system effects
Full-system, execution-driven simulation (PHARMsim)	Operating system, programs, input sets, disk images	Detailed, precise, accurate	Implementation complexity, simulation time overhead, correctness requirement

TABLE 1. Attributes of various performance modeling techniques.

lected, the cost associated with the disk space used to store the trace may be a limiting factor.

Prior work has argued that trace-driven simulation is no longer adequate for simulating modern, out-of-order processors (e.g. [1]). In fact, the vast majority of research papers published today employ execution-driven simulation and utilize relatively detailed and presumably precise simulation. A recent paper argued that precise simulation is very important and can dramatically affect the conclusions one might draw about the relative benefits of specific microarchitectural techniques [6]. Some of these conclusions were toned down in a subsequent publication [5].

One can reasonably conclude that the majority of recently published computer architecture research papers place a great deal of emphasis and effort on precision of simulation, and researchers invest large amounts of time implementing and exercising detailed simulation models.

In this paper, we show that conventional approaches to exercising processor simulators do so poorly vis-a-vis accuracy that, practically speaking, precision is unimportant. We argue that the correct approach is to build a simulator that is both precise and accurate. We accomplish this by building a simulator--PHARMsim--that does not *cheat* with respect to any aspect of simulation. Without the investment in such a simulator, we assert that it is impossible to determine whether or not the right abstractions and simplifications have been applied to either the simulator or the workload that is driving it.

We provide evidence that counters conventional wisdom for three factors that affect precision and accuracy in simulation. First, we show that operating system effects are important not just for commercial workloads (as shown by [8] and numerous others), but also for SPEC integer benchmarks [13]. Surprisingly, omitting the operating system can introduce error that exceeds 100% for benchmarks that, according to conventional wisdom, hardly exercise the operating system. Next, we show that simulating incorrectly predicted speculative paths is largely unimportant for both commercial workloads and SPEC integer benchmarks. In most cases, even with an aggressively speculative processor model that issues twice as many instructions as it retires, the bottom line effect on performance is usually less than 1%, and only 2.4% in the worst case. Finally, we argue that correct simulation of I/O behavior, even in uniprocessors, can affect simulator accuracy. We find that a direct-memory-access (DMA) engine that correctly models the timing of cache line invalidates to lines that are written by I/O devices can affect miss rates by up to 2% and performance by up to 1%, even in a uniprocessor with plenty of available memory bandwidth.

We have found that the main drawback of a simulator that does not cheat is the expense and overhead of correctly implementing such a simulator. For example, since our DMA engine implementation relies on the coherence protocol to operate correctly, the coherence protocol must be correctly implemented. Similarly, since the processors actually read values from the caches, rather than cheating by reading from an artificially-maintained flat memory image, DMA and multiprocessor coherence must be correctly maintained in the caches. Furthermore, within the processor core, the register renaming, branch redirection,

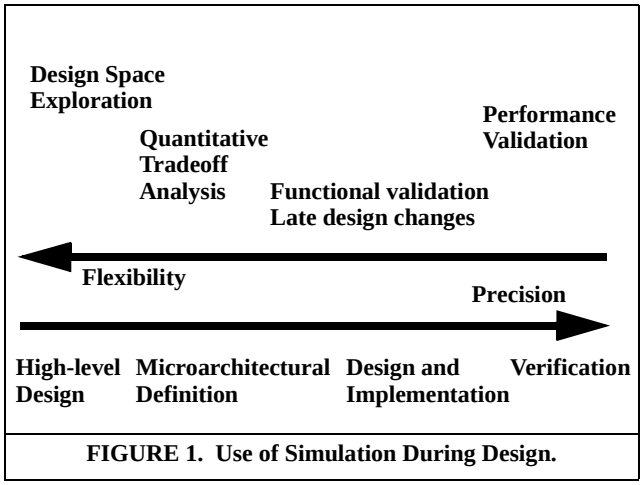


FIGURE 1. Use of Simulation During Design.

store queue forwarding, etc., must all operate correctly for the simulator to follow the correct path. Of course, this drawback is also an advantage: forcing a correctness requirement also forces us, as researchers, to be more thorough and realistic about the techniques that we propose, since we cannot "cheat" when we implement them in our simulator. Counter to our initial expectation, the simulation-time overhead of our simulator is surprisingly low, compared to competitive trace- or execution-driven simulators.

Section 2 provides further discussion on precision and accuracy and how they relate to processor simulation; Section 3 presents details of our PHARMsim simulation environment; Section 4 provides evidence for our three claims concerning operating system effects, speculative wrong-path execution, and DMA implementation; and Section 5 discusses conclusions and implications of our findings.

2.0 Flexibility, Precision, and Accuracy

As discussed already, design teams need simulators throughout all phases of the design cycle. As shown in Figure 1, the *precision* of a simulator tends to increase as the project proceeds from high-level design to later design stages. Here, we define *precision* as a measure of the fidelity of the simulated machine to the actual machine, as the machine is first envisioned and finally realized by its designers. The simulator's precision increases as a natural side effect of the needs of the designers; as the design itself is refined and more precisely defined, making quantitative design trade-offs requires a more precise simulator. Hence, additional features are added to the simulator to model these details. On some development projects, a separate performance simulator effectively disappears, and is replaced by simulation of register-transfer-level models expressed in a hardware-definition language (HDL).

As a consequence of increasing precision, flexibility in turn decreases. By flexibility we mean the ability of the simulator to continue to explore a broad design space. As more and more features are modeled precisely, it becomes increasingly difficult to support design space exploration that strays too far from the chosen direction. This trend also mirrors what is occurring in development; the further the project is from its initial concept, and the closer it is to final realization, the more difficult it becomes to make the major changes required by a broad change in the high-level design.

Besides flexibility and precision, there are several additional important attributes that characterize a simulator or simulation approach. These include simulation speed, functionality, usability, and *accuracy* of simulation. The *accuracy* of a simulator gauges its ability to closely model the real-world behavior of the processor or system being simulated, and manifests itself through simulated performance results that closely match the performance of the real system.

Accuracy is determined by two factors: again, by how closely the model matches the actual design (i.e. precision), but also by how the model is driven: how realistic is the "input" to the model? In general, analytical models and even fixed latency CPI equation models, as presented in Table 1, can be reasonably accurate, but are not very precise. Hence they are commonly used in industry, particularly for performance projections and competitive analysis, as well as early-stage feasibility and proof-of-concept analyses.

In contrast, academic researchers are prone to spend a great deal of time and energy building detailed simulation models that provide lots of precision, so that minute microarchitectural trade-offs can be studied thoroughly and exhaustively. Of course, the level of detail in an academic simulator must match the purpose of the study. For example, high-level limit studies are appropriately conduct with abstract and flexible models. On the other hand, detailed trade-off analyses must be made with fairly precise models. Some academic work exists that attempts to quantify simulator accuracy [2, 6].

Unlike precision, which can be quantified and rectified relatively early in the design cycle, accuracy is much more difficult to measure. Precision can be quantified by exercising both the performance simulator and progressive register-transfer-level realizations of the design with identical test cases, and the cycle-accurate results can be compared and reconciled to correct either the simulator or the design. This is in fact a natural side effect of the performance validation that should occur during a properly managed design cycle.

However, accuracy cannot be so easily determined, since accuracy depends not only on the simulator's precision,

but also on how closely the inputs to the simulation match the real-world environment in which the system being designed will ultimately operate. It is usually considered extremely difficult to recreate these circumstances in such a way that they can be used to drive a detailed performance simulator.

The initial work on full-system simulation from the Stanford SimOS project [10] established that this is indeed possible. However, the complexities of doing so have effectively deterred the majority of the research community from adopting full-system simulation into their repertoire. We emphatically agree with other proponents of full-system simulation and argue that all architecture researchers should seriously consider adoption of full-system simulation, despite the up-front cost of doing so. The evidence in Section 4 strongly supports the assertion that simulators that ignore system effects, no matter how precise, are likely to be so inaccurate as to be useless, even for CPU intensive benchmarks like SPECINT 2000.

In practice, the accuracy of a performance simulator is usually not evaluated until it's "too late," that is to say after hardware is available and stable enough to boot an operating system and run real workloads. At this point, due to several generations of software changes and numerous potentially compensating errors, it becomes very difficult to precisely quantify the accuracy of a performance model. Furthermore, from a practical standpoint, doing so is only useful from an academic and quality assurance viewpoint, and is not driven by immediate design needs. Hence, at least in our experience, such an evaluation is either performed poorly or not at all.

3.0 PharmSim Overview

We have constructed a PowerPC-based simulation infrastructure using the SimOS-PPC and SimpleMP simulators. SimOS is a complete machine simulation environment consisting of simulators for the major components of a computer system (cpus, memory hierarchy, disks, console, ethernet) [10]. We use a version of SimOS which simulates PowerPC-based computer systems running the AIX 4.3 operating system [7]. SimpleMP is a detailed execution-driven multiprocessor simulator that simulates out-of-order processor cores, including branch prediction, speculative execution and a cache coherent memory system[9] using a Sun Gigaplane-XB-like coherence protocol [4]. Integrating the SimpleMP simulator into SimOS required significant changes to SimpleMP in order to accurately support the PowerPC architecture. In this section, we discuss these modifications.

SimpleMP was missing much of the functionality necessary to support system level code, in both the processor core and memory system. We augment SimpleMP with

Processor Parameters	
decode/issue/commit width	8/8/8
RUU/LSQ size	128/64
Functional Units	8 Int ALUs, 3 Int Mult/Div, 3 FP ALUs, 4 FP Mult/Div 3 LD/ST Ports
Branch Predictor	Combined bimodal (8k entry)/ gshare (8k entry) with 8k choice predictor, 8k 4-way SA BTB, 64 entry RAS
Memory System	
L1 I Cache (latency)	64K 2 way set associative (1 cycle)
L1 DCache (latency)	256K 4 way set associative (1 cycle)
L2 Unified Cache	4MB 4 way set associative (10 cycles)
blocksize (all caches)	64 bytes
DRAM latency	70 cycles

TABLE 2. Simulation Parameters.

support for all of the instructions (system-mode and user-mode) in the PowerPC instruction-set architecture. For some of the relatively complex PowerPC instructions (e.g. load/store string instructions) we use an instruction-cracking scheme similar to that used in the POWER4 processor which translates a PowerPC instruction into several simpler RISC-like operations [14]. We also augment the processor core with support for precise interrupt handling and PowerPC context-synchronizing instructions (e.g., isync, rfi).

The SimpleMP memory system required major changes in order to support unaligned memory references, PowerPC address translation, and the set of PowerPC cache management instructions. To handle unaligned memory references (which are allowed in the PowerPC architecture) the processor core splits each unaligned memory reference that crosses a cache block boundary into two smaller aligned references which are then each issued to the SimpleMP memory system.

In order to accurately model PowerPC virtual memory hardware, we were forced to implement a PowerPC memory management unit (MMU) from scratch, including a translation lookaside buffer (TLB), TLB refill mechanism, and reference and change bit setting hardware. On a TLB miss, we simulate a hardware TLB miss handler which walks the page table by issuing memory references to the

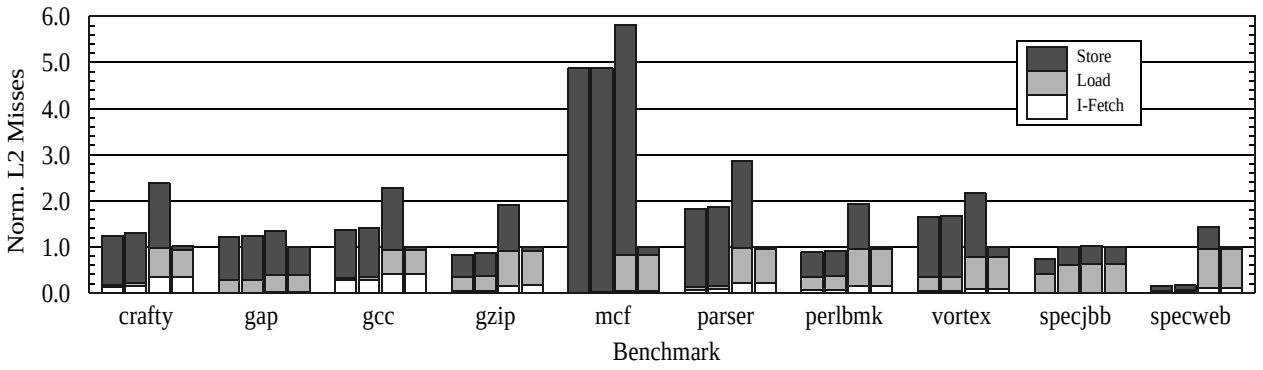


FIGURE 2. Cache effect of simulating the whole system.

The stacked bars show misses to an 8MB, 4-way set-associative cache with 64B lines due to instruction fetches, loads, and stores, normalized to the rightmost case where the whole system is simulated. The leftmost case shows the effects of program references only. The second case adds shared library code, the third case adds operating system code, and the rightmost case adds cache control instructions issued by the operating system. The worst-case error, in MCF, is 5.8x.

simulated memory hierarchy. In the event of a memory management exception (e.g., page fault, protection exception), the MMU signals the processor which traps to the appropriate OS exception handler. The MMU also maintains and updates a page's reference and dirty bits by issuing single-byte stores to the simulated memory hierarchy when a page whose reference or change bit is not set is first referenced or written.

The PowerPC architecture includes many cache management instructions (e.g. data cache block invalidate, data cache block zero, etc.) which are used in both system and user-level code. Implementing each of these instructions required significant changes to the SimpleMP coherence protocol.

We also augment the SimpleMP memory system to support coherent I/O. Both SimpleMP and SimpleScalar [3] perform I/O "magically" by proxying system calls and instantaneously updating a cache's contents to reflect the new memory contents. Obviously, this mechanism does not accurately model how I/O is performed in real systems. To accurately model coherent I/O, we added support to SimOS and SimpleMP for I/O controllers to initiate DMA transfers into memory and invalidate the corresponding blocks in each processor's caches.

For all of the data presented in this paper, we use the machine configuration summarized in Table 2.

4.0 Sources of Inaccuracy

In this section, we study three possible sources of inaccuracy in processor simulation, and quantify their effects on a set of SPECINT2000 and commercial server workloads. The three case studies are the effect of operating system paths, the effect of direct memory access (DMA) transfers caused by disk input/output (I/O), and the effect of speculatively executed incorrect branch paths.

We are able to study these effects in detail only because we have implemented a simulator that does not cheat. The vast majority of prior simulation work either assumes that these effects are insignificant, and fails to consider them, or, assuming the opposite, do implement them but do not quantify the necessity of this implementation overhead.

We present these three case studies to examine if and how inaccuracy is introduced into simulation, and to quantify how relatively important each of these effects is. Current practice in the architecture research community focuses lots of effort on wrong-path execution, and arguably trades off investment in the other two factors to capture wrong-path behavior.

4.1 Operating System Effects

The first effect we study has been examined at length in prior work, particularly for commercial workloads that spend a nontrivial fraction of execution time in the operating system (e.g. [8]). However, conventional wisdom holds that the SPECINT2000 benchmarks spend very little time in the operating system, and can be safely modeled with user-mode instructions only. Our experience with PHARMsim, however, has shown that this is a fallacy. Our detailed simulations have shown that ignoring the effects of operating system instructions can lead to errors of 100% or more when executing SPECINT2000 benchmarks. A detailed analysis of these findings is beyond the scope of this paper, and is left to future work. However, we do present evidence here that helps explain this unanticipated source of error.

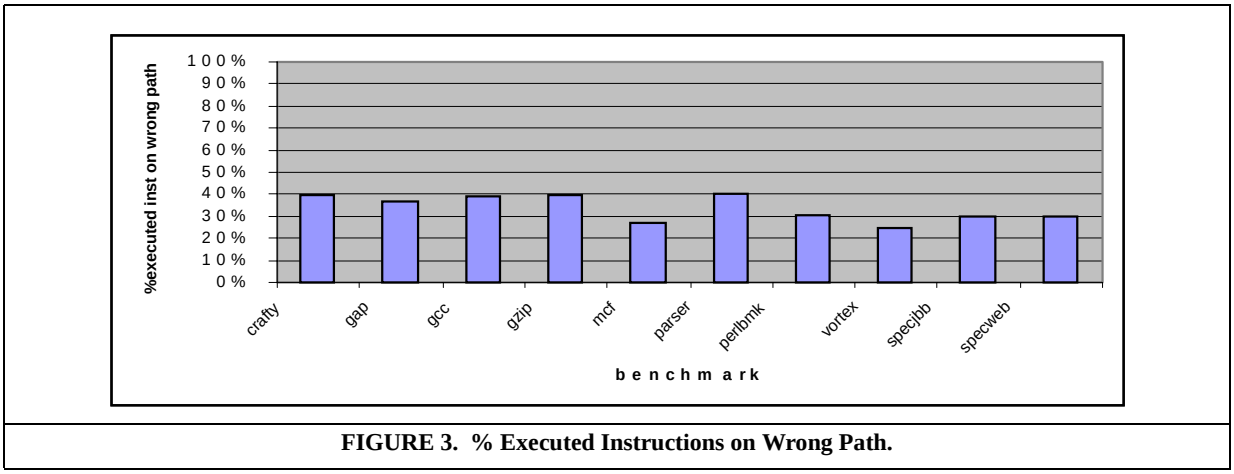


FIGURE 3. % Executed Instructions on Wrong Path.

Figure 2 summarizes off-chip memory traffic for the set of benchmarks we studied, which includes eight of the SPECINT2000 benchmarks and two multiuser commercial workloads, SPECWEB99 and SPECJBB2000 [13]. The stacked bars show misses from an 8MB, 4-way set-associative cache with 64B lines. Miss rates are normalized to the correct case, where all references generated by both the user-mode program, shared libraries, operating system, and special cache control instructions are properly accounted for. The four stacked bars for each benchmark show, from left to right, the effects of the user program only, additional effects from shared library code, additional effects from all of the operating system, and, in the rightmost case, additional effects from special PowerPC cache control instructions issued by the operating system.

These four cases roughly correspond to simulation approaches used in the past: the leftmost case to trace-collection schemes like Atom [12] that instrument and trace user programs only (later versions of Atom handled shared libraries and even parts of the operating system); the second case to a tool like Simplescalar that requires statically-linked objects and then performs system-call translation [3]; the third case to a tracing technique that captures all loads, stores, and instruction fetches by recording off-chip bus signals, but fails to capture explicit cache control references that avoid such signals; and finally, a full-system simulator like PHARMsim that captures and correctly models all references that affect the cache.

In all cases, we see that more references are captured, and additional misses generated, as we proceed from modeling only the user program's references to modeling all of the operating system. In the worst case (*perlbmk*), the number of misses more than doubles, and is significant even in the best case (*specjbb*).

However, interestingly enough, adding the effects of special cache control instructions reverses this direction, and dramatically reduces the number of misses that the cache model encounters. This behavior is caused by the AIX operating system's aggressive use of the PowerPC dcbz (data cache block zero) instruction. This instruction writes an entire cache line with zeroes. Aggressive hardware implements this instruction by avoiding an off-chip memory reference even when it misses the cache, since the whole line will be overwritten by the instruction. This instruction is similar to the wh64 (write hint 64) instruction in the Alpha instruction set [11].

We have analyzed the use of dcbz in the AIX operating system, and have found that the page fault handler issues a series of dcbz instructions that span the entire 4K virtual memory page whenever a program page faults on a new, previously unmapped page. This is a legitimate optimization, since a newly mapped page cannot contain any valid data. Hence, the operating system can safely zero out the page before returning to the user. As a side effect, the program can avoid cache misses to newly referenced pages, since the dcbz instructions directly install those lines into the cache.

The effect of the dcbz instructions is particularly pronounced for the *mcf* benchmark, which allocates and initializes tens of megabytes of heap space to store its internal data structure. Naive simulation, whether of the user program only, or even including operating system effects, can dramatically overstate—to the tune of 5.8x—the number of cache misses encountered by this program. Note that virtually all of the store misses, which dominate the memory traffic of this benchmark, disappear when the dcbz instructions are correctly modeled. There is a similar, though less pronounced, trend for all of the other workloads except *specjbb*. We attribute specjbb's behavior to the fact that we are capturing a snapshot of steady-state execution for this benchmark, rather than end-to-end program execution. As a result, memory has already been allocated and initialized, and the AIX page fault handler does not issue *dcbz* instructions.

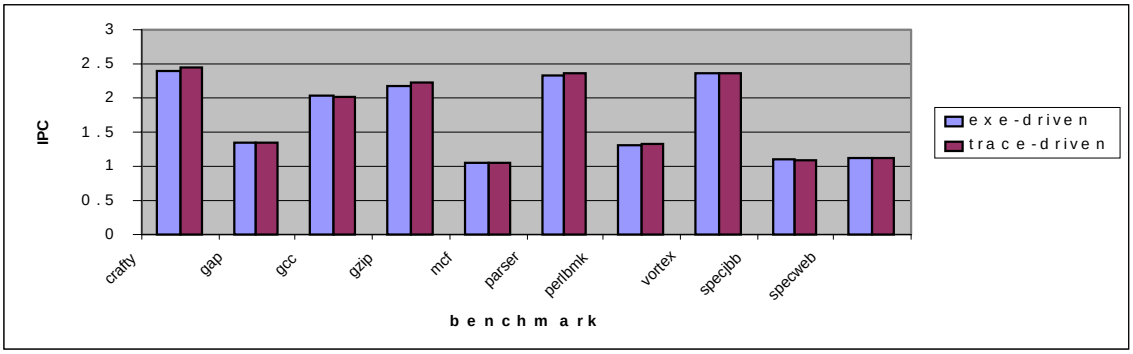


FIGURE 4. Effect of wrong-path instructions on IPC.

We suspect that other operating systems have similar optimizations in their page fault handlers, though we currently have no way of verifying this fact. However, the fact that a detailed, precise, simulator described in a recent study [6] reflected a roughly 20% error on many real benchmarks leads us to believe that such optimizations are widespread and dramatically impact the accuracy of simulation that ignores operating system references.

4.2 Direct Memory Access (DMA)

High-performance systems use some form of autonomous, coherent, I/O agent to perform I/O requests in the system using DMA. This mechanism allows the processor to continue performing other tasks (for example, run another ready process) while the I/O request serviced. Most current simulators either handle I/O through system call proxy or perform the I/O into a flat memory image. Performing flat-memory I/O is functionally correct in simulators which do not include values in cache models, because all values are obtained from the flat memory. However, using this technique the state of cachelines accessed by the I/O agent can be incorrect in one of two ways: 1) In the case of a DMA write (i.e. write to memory, read from I/O device), the line should be marked invalid in the processor's cache; 2) In the case of a DMA read (i.e. read from memory, write to I/O device), the line should become shared.[1]

In execution driven simulators which track memory values throughout the hierarchy (i.e. SimpleMP) I/O causes an additional problem. Since multiple coherence transactions pertaining to regions of memory subject to I/O can be in flight (i.e. a cache block may be in a pending state in multiple caches at the time of an I/O request), the system must be quiesced (i.e. all processor and coherence activity must stop) to effectively flatten the memory image. The I/O can then be performed with any updated values copied (magically, bypassing the performance model) into processors' caches to maintain a consistent view of memory.

1. Provided DMA read requests are handled as regular read requests.

In PHARMsim, in order to determine the effect of cache state errors due to disk I/O activity, as well as system quiescing, we have added an I/O agent (DMA engine) which actually performs the necessary coherence transactions and data transfers between the coherence network/cache model and the disk. This approach avoids the aforementioned inaccuracies and additionally contributes realistic contention on the address and data networks due to disk I/O. Without building this model, we cannot know whether neglecting these I/O effects maintains our stated goals of both simulator accuracy and precision.

We found that for the benchmarks in Figure 2, the I/O effects are small. The only benchmark showing a non-trivial change in cache hit rate is a version of *mcf* in which we have artificially constrained the physical memory size to 64MB to force paging (the L1-I, L1-D, and L2 cache hit rates are reduced by 1.5%.) If we increase the available physical memory to eliminate paging, because AIX implements the *dcbz* optimization mentioned previously (Section 4.1) for newly allocated pages, all I/O coherence events in *mcf* are eliminated. We believe the relative insensitivity to I/O effects occurs due to the effectiveness of disk caches (the benchmarks shown have a paltry number of I/O coherence events compared with other coherence events) and also the nature of the benchmarks--which are not meant to stress I/O performance. We also point out that we expect the execution time difference for single-programmed workloads, provided I/O latency is modelled, to be small given the large disparity between I/O and coherence latency.

One might expect the multi-programmed commercial workloads (*specjbb* and *specweb*) to have required disk activity due to database logging or increased working-set sizes common to commercial applications. However, *specjbb* has no database component and *specweb* has less than 1% of coherence transactions due to I/O in our snapshots, leading to negligible I/O effects.

In order to stress the I/O subsystem, we created our own

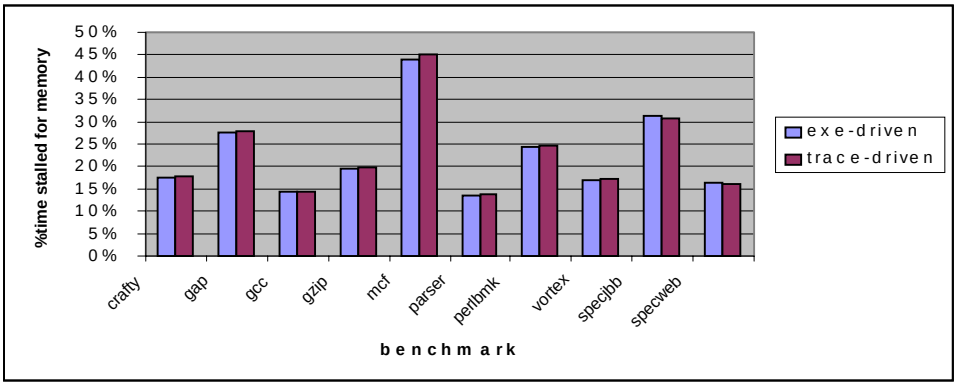


FIGURE 5. Effect of wrong-path instructions on memory stall time.

multi-programmed workload with a combination of file I/O (reading multiple uncached files using the unix 'cat' command) and computation (using gzip of a simulator source code file) and measured the execution time to complete the workload end-to-end. We measured up to a 2.5% reduction in cache hit rate, and a 1.1% increase in execution time when I/O traffic was modelled. We also determined the effect of quiescing the system for I/O related events was less than 0.9% on overall execution time.

Even for this I/O intensive workload (4.8% of coherence traffic due to I/O), the overall effect of I/O on simulation accuracy and precision is small, but measurable, largely due to low contention in the coherence network (over 97% of all coherence transactions occur with fewer than two transactions already outstanding, with 16 possible in our network). However, in larger-scale systems with more processors, we expect greater contention in the coherence and data networks due to three things: 1) Additional demand traffic from other processors; 2) Increased I/O requirements for supplying them; and 3) Difficulty in scaling coherent interconnect in large systems. Evaluating this space is beyond the scope of this work.

4.3 Effect of Wrong-Path Execution

In this section we quantify the impact of ignoring wrong-path instructions on a simulator's results. To perform this comparison, we execute each benchmark on two versions of PHARMSim: the standard execution-driven version, and a modified "trace-based" version which uses a perfect branch predictor to throttle the PHARMSim fetch stage when the machine would ordinarily be fetching an incorrect branch path. Using these two configurations we can evaluate the impact of wrong-path instructions on final performance results.

Figure 3 shows the percentage of instructions reaching the execution stage of the pipeline that are on a mispredicted branch path. For this machine configuration, between 25% and 40% of all instructions executed are on the wrong path. Because this percentage is so high, one would intuitively expect the side-effects of wrong path instructions to have a significant effect on total execution time. However, as can be seen in Figure 4 this is not the case.

Figure 4 shows the difference in IPC as measured by the trace-driven version of PHARMSim and the execution-driven version of PHARMSim for each of the benchmarks. We see that the difference in IPC reported by the two simulators is only 0.97% on average. In the worst case, the benchmark crafty, the difference is only 2.4%. The marginal differences in IPC are the caused by the interaction of wrong-path instructions with non-speculative instructions through the cache hierarchy and branch-predictor.

Wrong-path memory operations may have a positive or negative effect on overall performance. If a wrong path memory operation touches a cache block that the correct execution path will touch in the future, the wrong path instruction may have a prefetching effect, reducing the number of memory related stall cycles for the program. However, if a wrong-path instruction touches a cache block that will not be used in the immediate future, the speculative memory reference may pollute the cache or may compete with subsequent correct-path memory operations for memory system bandwidth. The PHARMSim TLB does not service a TLB miss until the instruction which caused the miss is non-speculative, which in effect filters some of the wrong-path memory references from the cache hierarchy. Figure 5 shows the effects of wrong-path instructions on memory system stall time. For all of the benchmarks except SPECjbb and SPECweb, the inclusion of wrong-path instructions has a positive effect on memory stall time. In all cases, however, the effect is very small. On average, there is only a 0.3% difference between the total number of memory stall cycles when executing with and without wrong-path instructions.

The execution of wrong-path instructions may also affect

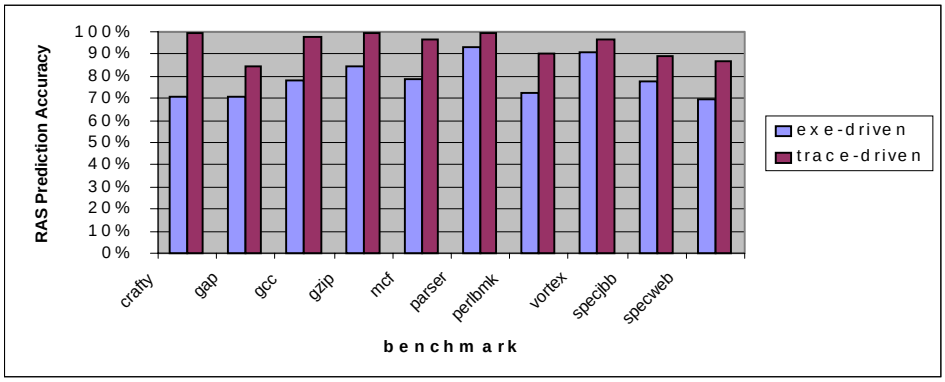


FIGURE 6. Effect of wrong-path instructions on return-address stack accuracy.

branch predictor performance. The branch predictor used for the results in this section is a combining predictor whose pattern history tables are not updated speculatively, and whose branch history register is checkpointed before each branch and restored in the event of a branch misprediction. Consequently, the main branch predictor is not polluted by wrong-path instructions and suffers no performance degradation. However, the 64-entry return address stack is updated speculatively and is not recovered in the event of a branch misprediction. As shown in Figure 6 the pollution of the return address stack by wrong path instructions significantly affects its performance. On average, the accuracy of the return address stack decreases by 15%. In the worst case (crafty) accuracy is reduced by 29%. Despite this RAS performance loss, overall performance results are not barely affected, as shown in Figure 4.

Overall, ignoring the effects of wrong-path execution has almost no impact on performance. Consequently, we believe trace-driven simulation without wrong-path instructions is a valid method for estimating uniprocessor performance. Execution-driven uniprocessor simulators may also ignore wrong path instructions to improve simulation efficiency. Although these results validate the use of trace-driven simulation to evaluate single-threaded workloads, the performance of multi-threaded workloads which include communication among threads should not be evaluated using trace-driven simulators for reasons which are beyond the scope of this paper.

5.0 Conclusions

This paper studies three factors that can affect the accuracy of uniprocessor simulation: operating system effects, direct memory access by I/O devices, and wrong-path speculative execution. Using PHARMsim, a detailed full-system simulator that does not cheat, we are able to show that operating system references should be fully modeled, even for benchmarks like SPECINT2000 that have historically been considered safe for user-mode-only simulation. In the case of the AIX operating system, this is due to optimizations in the page fault handler that employ explicit cache control instructions to avoid unnecessary cache misses. Further, we find that correct modeling of DMA traffic can have a nontrivial effect on performance, and should be accounted for in workloads that perform a significant amount of I/O. Finally, we show that wrong-path speculative execution has a nearly indiscernible effect on overall performance. Though individual microarchitectural structures like the return address stack can be negatively affected by these paths, the overall contribution of these effects is so minimal that ignoring speculative paths is safe for the workloads we study.

Our study is far from complete, as there are numerous other effects we are currently studying and plan to report on in the future. These include the effects of speculative and non-speculative TLB refills, more detailed analysis of branch predictor updates, evaluation of DMA transfers directly into the cache hierarchy, etc. However, we do make the following conclusions and suggestions based on the evidence presented herein:

- All detailed processor simulations, even if only running SPECINT-like benchmarks, should fully account for operating system references. The extreme errors that are introduced if these effects are not modeled make any such simulations so inaccurate as to be meaningless. This conclusion should have a significant impact on the research community and the peer review process.
- Trace-based simulation, which is still widely used in industry, should not be dismissed in favor of execution-driven simulation. Our evidence suggests that traces that include operating system references but omit wrong-path speculative references are far more useful than execution-driven simulation of user-mode programs.

- Though demonstrably more accurate than execution-driven simulation, trace-based simulations may still have shortcomings that make execution-driven simulation attractive. For example, most hardware tracing schemes are incapable of capturing register or memory values. Hence, study of techniques that exploit value locality is not possible with such traces.
- Workloads that perform a nontrivial amount of I/O should be simulated in a way that properly accounts for the additional memory traffic induced by DMA transfers. Without such an accounting, simulation results may not be acceptably accurate. On the other hand, workloads like SPECINT2000, with minimal I/O, can safely be simulated without accurate modeling of DMA effects.

Finally, we want to point out that lack of proper simulation infrastructure should not serve as a valid excuse for avoiding both precise and accurate processor simulation, and the research community as a whole needs to accept this fact. Our research group has made a significant investment in simulation infrastructure that also builds heavily on work done by others. The fact that we have been able to develop this infrastructure serves as an existence proof that it is possible, even with the limited means available within academia.

6.0 Acknowledgments

Many individuals have contributed to the work described in this paper. Among them are current and former members of the PHARM research group at the University of Wisconsin, Ravi Rajwar who wrote the SimpleMP simulator, Pat Bohrer and others at IBM Research who ported SimOS to the PowerPC architecture, as well as the original authors of the SimOS toolset at Stanford. We are heavily indebted to all of these individuals. This work was also supported by donations from IBM and Intel and NSF grants CCR-0073440, CCR-0083126, and EIA-0103670.

References

[1] Bryan Black, Andrew S Huang, Mikko H. Lipasti, and John P. Shen. Can trace-driven simulators accurately predict superscalar performance? In *Proceedings of the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '96)*, October 1996.

[2] Bryan Black and John Paul Shen. Calibration of microprocessor performance models. *Computer*, 31(5):59–65, May 1998.

[3] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin Computre Sciences, 1997.

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

[5] R. Desikan, D. Burger, S.W. Keckler, L. Cruz, F. Latorre, A. Gonzalez, and M. Valero. Errata on measuring experimental error in microprocessor simulation. Computer Architecture News, March 2002.

[6] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA-01)*, June 2001.

[7] Tom Keller, Ann Marie Maynard, Rick Simpson, and Pat Bohrer. Simos-ppc full system simulator. http://www.cs.utexas.edu/users/cart/simOS.

[8] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *ACM SIG-PLAN Notices*, 29(11):145–156, November 1994.

[9] Ravi Rajwar and Jim Goodman. Simplemp multiprocessor simulator. Personal communication., 2000.

[10] Mendel Rosenblum. Simos full system simulator. http://simos.stanford.edu.

[11] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Maynard, MA, 1992.

[12] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[13] Systems Performance Evaluation Cooperative. SPEC benchmarks. http://www.spec.org.

[14] Joel M. Tendler, S. Dodson, S. Fields, and B. Sinharoy. IBM eserver POWER4 system microarchitecture. IBM Whitepaper, October 2001.

New Challenges in Benchmarking Future Processors

Shubhendu S. Mukherjee
Intel Corporation
Shubu.Mukherjee@intel.com

ABSTRACT

The advent of system on a chip, fault-tolerant features, and multiple power modes in the mainline processor market has significant impact on how we would benchmark future processors. Unfortunately, only a subset of these modes may provide the highest performance for these chips. If vendors report performance numbers only for these highest performing modes, then customers are faced with the challenge of selecting the specific processor and the specific configuration for his or her operating environment without the benefit of any comparable benchmark numbers.

In this paper I examine three different categories of hardware modes that a processor chip could be configured in. These are the use of a snoopy or directory protocol, the presence or absence of fault tolerance, and presence of different power modes. Based on these examples, I classify the modes into performance-centric and environment-centric modes and propose that vendors report performance numbers for these different modes. This would allow customers to compare processor chips from different vendors under different operating conditions.

1. INTRODUCTION

Four technology trends—exponential proliferation of on-chip transistors, the constant RC delay of long on-chip wires, transient faults due to cosmic ray strikes, and power constraints—may necessitate changes to the way we would benchmark future processors. The exponential proliferation in the number of on-chip transistors—fueled by Moore's Law—is forcing designers to find innovative ways to use these transistors to remain competitive in the high-performance processor market.

Unfortunately, the speed of long on-chip wires is remaining constant due to the constant RC delay, unlike transistors that speed up by roughly 30% with every new technology generation. This trend, coupled with the complexity of dealing with such an enormous number of transistors, is forcing designers to refrain from building a big, complicated, and monolithic uniprocessor on a single chip. Rather, the trend is towards Chip Multiprocessors (CMPs) with many processors on the same chip (e.g., IBM Power4 [1], HP Mako [2]), and/or processors with large caches that can use up the available number of on-chip transistors.

Thus, greater amounts of system functionality, such as multiple processors and large caches, are getting integrated on the same chip. Additional system components, such as memory controllers, multiprocessor network routers, and cache-coherent directory controllers, have also started migrating to the same die as the processor chip (e.g., Alpha 21364 [3]). Clearly, we are seeing the emergence of *system on a chip* in the mainline processor market.

Unfortunately, while most such on-chip functions improve performance for specific application domains, two other trends have begun limiting the performance of these processors. First, soft errors due to cosmic ray strikes promise to increase dramatically in the next few generations. This is arising because of the proliferation of number of on-chip transistors and the reduction in voltage levels of the chips. Consequently, designers will be forced to use some of the on-chip transistors for ECC, parity, or other forms of error detection and/or correction. Other mechanisms, such as lockstepping two complete processors (e.g., as in Compaq Himalaya [4] or IBM G5 [5]) or the use of redundant multithreading (e.g., AR-SMT [7], DIVA [6], SRT [8], or CRT [9]) to detect faults, may also become popular with CMP designs. Additionally, support for transparent fault recovery in hardware, such as tri-modular redundancy [10], may necessitate even more sophisticated hardware support, which may further limit a chip's performance.

Second, processors are facing severe limits on average and peak power dissipation [11]. A greater number of on-chip transistors demands greater power consumption (and consequent heat dissipation), if we follow the same design methods as in the past. So, designers must carefully use the on-chip resources to

reduce wasted work. Additionally, designers may have to slow down processors, either statically or dynamically, if they consume too much power or completely shut some of them off to balance the total power dissipation in a CMP chip.

The above trends have significant implications on the way we would benchmark future processors and systems. Currently, two of the most popular benchmark suites are the SPEC CPU suite and the TPC suite. The SPEC CPU 2000 suite (http://www.spec.org) measures a processor's integer, floating point, and memory system performance using a set of 25 benchmarks. SPEC allows us to characterize uniprocessor performance using two speed metrics—*SPEC base* and *SPEC peak* performance. Base performance measures the performance of a processor with a set of fixed and uniform compiler flags applied to all benchmarks in the suite. In contrast, peak performance measures the performance of a processor with benchmark-specific compilation flags. Similarly, SPEC allows us to characterize multiprocessor or multithreaded performance using *SPEC rate*, which measures the rate at which one or more processors may complete multiple copies of the same benchmark. SPEC rate can be similarly categorized into base and peak rates.

TPC (http://www.tpc.org) consists of a suite of benchmarks to measure the performance of a system running transaction processing and database workloads. Performance measurements can be reported as absolute performance or in terms of price/performance. For example, performance measurements for the TPC-C benchmarks are reported as both *tpm* (transactions per minute) and *price/tpm*.

Unfortunately, neither the SPEC nor the TPC suite was designed to measure a system-on-a-chip's performance. The key problem is that these systems on a chip could potentially have optional hardware modes that will be configured in different ways. For example, to deliver high performance on TPC-C like benchmarks, a vendor may statically configure the system on a chip to use a snoopy cache coherence protocol that provides fast cache-to-cache transfers between processors. In contrast, the same vendor may configure the chip to use a directory-based coherence protocol if they incorporate these chips in a large, scalable, multiprocessor system (Section 2). Similarly, the inclusion of fault detection and/or recovery can dramatically reduce the performance of a system on a chip when the fault detection mode is turned on (Section 3). Finally, systems that will be configured for lower power dissipation may switch to a specific static mode—for example, by turning off several processors in a CMP—or reduce the frequency of the entire chip itself (Section 4).

Neither the SPEC nor the TPC suite requires vendors to benchmark their processors or systems in all such possible configurations. Thus, a vendor may report its performance numbers in its highest performing modes—with benchmark suite-specific system configuration, no fault detection, and highest power dissipation.

Unfortunately, such reporting can both be confusing and unrealistic. It can be confusing because, for example, a customer may have no clue how to compare systems-on-a-chip for processors with fault detection and recovery enabled. The customer may desire to buy a highly reliable computing system, but the only numbers he/she may find for comparison are numbers for these systems in the non-fault detection mode.

It can be unrealistic because, for example, vendors can potentially produce results in an environment (e.g., in a basement or in a building with thick concrete) that could have reduced effect of cosmic rays. Such measurements are of little value to a customer who wants to use systems in operating environments (e.g., in an office with large glass windows or in a airplane) where the effect of cosmic rays may be quite high.

Based on the above discussions, I examine potential solutions in Section 5. We can divide these solutions into two broad categories based on the nature of the hardware modes. These two modes are performance-centric modes and environment-centric modes. Performance-centric modes are those that configure the system in specific hardware modes for specific benchmarks and benchmark suites. One possible solution for such modes is to use the SPEC characterization of base and peak. The base mode could be one mode specified by the system on a chip for all configurations. However, the peak mode can be expanded to include benchmark-specific hardware modes, beyond the benchmark-specific compiler flags that are already in use.

Environment-centric modes are those that specify and evaluate benchmarks based on the specific environment the system-on-a-chip is embedded in during measurements. Both fault detection and power modes fall under this category. At the minimum, systems must specify in what environment the measurements were done. It would be even better if systems could specify the cosmic ray flux and the maximum power dissipation possible in that environment (although gathering such data can be quite

expensive [12]) as well as multiple numbers for different points in the environment spectrum, such as high and low cosmic ray flux and high and low power dissipation modes.

2. SNOOPY VS. DIRECTORY PROTOCOLS

Most shared-memory multiprocessors use a coherence protocol to keep per-processor caches coherent. Typically, a cache block in a processor's cache in a cache-coherent multiprocessor system has at least three basic states—invalid, shared, and exclusive. Usually, a cache block becomes shared when a processor retrieves the cache block in read-only state (possibly resulting from a cache miss from a load instruction). Similarly, a cache block becomes exclusive when a processor retrieves a cache block in writeable state (possibly resulting from a cache miss from a store instruction).

The two most common coherence protocols are snoopy and directory protocols, which differ in how these states are manipulated. A snoopy protocol usually relies on a broadcast mechanism, such as a shared bus, to facilitate the state transitions. To request a shared block a processor sends snoop requests to all processors and memories that can have that block. Either a processor's cache or a memory returns the block to the processor in the shared state. A similar sequence of events occurs when a processor requests an exclusive block. Unfortunately, snoopy protocols do not scale well to large systems. This is either because the shared broadcast medium may not scale or because expensive broadcast messages must be sent on most processor cache misses.

Directory protocols allow cache-coherent shared-memory multiprocessors to scale to a large number of processors by avoiding a broadcast medium and broadcast messages. Instead, to request a shared block, a processor sends its request to a pre-selected "home" node. Typically, the home node would return the data. However, different variations in this strategy are possible. To request an exclusive block, a processor sends its request to the home node. The home node tracks down one or more sharers of the block, invalidates the cache blocks in all these sharers, and then sends the response back to the requester. Alternatively, the home node can also request one of the sharers to forward the block to the original requestor.

Although directory protocols allow systems to scale well, they are not optimized for producer-consumer or migratory sharing patterns in which a processor writes data that may be needed by another processor in the near future. For snoopy protocols, a processor can obtain the data in two hops: one for the request message and the second one for the response message. However, for directory protocols, this can take three hops: the first one for request to the home node, the second for the directory to forward the request to the processor with exclusive access to the block, and finally the third one for the response to propagate back to the requestor.

Thus, a directory protocol penalizes sharing patterns that require cache-to-cache transfers, particularly for small-scale systems that can benefit from a snoopy protocol. This can have a significant impact on the performance of benchmarks, such as TPC-C, that are dominated by such cache-to-cache transfers. For example, researchers have estimated that such cache-to-cache transfers could account for more than 50% of L2 cache misses in TPC-C like benchmarks [13][14]. Worse, the impact of such cache-to-cache transfers rises as cache sizes increase. This is because a bigger cache reduces other kinds of misses, such as conflict and capacity misses, but does not significantly reduce sharing misses that cause cache-to-cache transfers. Thus, small-scale commercial systems, such as the IBM Northstar [16], rely on snoopy protocols to provide aggressive performance on TPC-C like benchmarks. Recently, Martin, et al. [15] have shown that it is possible to build a protocol that adapts between a snoopy and directory protocol. Such adaptive protocols may work for large systems. It is not clear whether such adaptive protocols are a winner in price/performance for small-scale systems.

This dichotomy between snoopy and directory protocols in the performance and scalability spectra makes it challenging for vendors to build processor chips with full or partial support for the protocol on the chip. Typically, vendors would like to optimize systems that sell the most. And, systems that sell the most are small-scale systems that can be built with snoopy protocols. However, vendors also like to support large systems because they typically provide higher profit margins and assure a customer of a scalable system. Consequently, vendors are likely to end up supporting both a snoopy and a directory protocol on the same chip. In the past, such a dichotomy was not a problem because the protocol controllers would be off-chip; vendors would manufacture different chip sets for systems of different sizes.

With the advent of this new generation of processors, a customer must potentially deal with different hardware modes for small-scale and large- scale systems and ensure that the system he or she is buying reflects the

benchmark numbers for the appropriate mode he or she is interested in. Unfortunately, today vendors do not have to publish numbers for these different modes for the same processor.

3. FAULT DETECTION AND RECOVERY

Support for fault tolerance—that is, fault detection and recovery mechanisms—in the mainline processor market may also force vendors to create different hardware modes. This is because today's microprocessors are vulnerable to transient hardware faults caused by alpha particle and cosmic ray strikes.

Strikes by cosmic ray particles, such as neutrons, are particularly critical because of the absence of any practical way to protect microprocessor chips from such strikes. As individual transistors shrink in size with succeeding technology generations, they become less vulnerable to cosmic ray strikes. However, decreasing voltage levels and exponentially increasing transistor counts cause overall susceptibility of a chip to increase rapidly. Further, the impact of cosmic ray strikes grow by orders of magnitude in an airplane or on higher elevations.

To compound the problem, achieving a particular failure rate for a large multiprocessor server requires an even lower failure rate for the individual microprocessors that comprise it. Due to these trends, fault detection and recovery techniques, currently used only for mission-critical systems, would very likely become common in all but the least expensive microprocessor devices.

Unfortunately, current known techniques for fault detection and fault recovery degrade a chip's performance quite dramatically. Traditionally, processors have used error detection and/or correction codes (e.g., parity, ECC, CRC) to meet failure specifications. However, such error detection and correction codes do not cover logic blocks effectively. Additionally, error codes may degrade a pipeline's performance by adding extra pipeline stages**.**

Consequently, vendors may have to resort to other techniques for fault detection and recovery. There are three other known techniques for fault detection: lockstepping, instruction recycling, and redundant multithreading. In lockstepping, two complete and identical processors run in lockstep—performing the same computation in every cycle. Inputs to both processors are replicated in each cycle. Similarly, in each cycle outputs from both processors are compared for mismatch. On a mismatch, the hardware typically initiates a hardware or software recovery sequence. The advent of Chip Multiprocessors with two or more processors on the same chip would enable such lockstep checking on the chip itself. Because lockstepping uses two completely separate processor cores, it can catch both permanent faults (e.g., from electron migration) and transient faults (e.g, from cosmic ray strikes).

Instruction recycling is another technique for fault detection. Instead of comparing instructions from different processors, instruction recycling runs the same instruction twice through the same processor and checks the corresponding outputs from these instructions for mismatch. Since instruction recycling runs copies of the same instruction through the same processor, it could potentially catch all single transient faults, but not all permanent faults.

Finally, there is a new class of fault detection techniques that can be classified under Redundant Multithreading (RMT). RMT runs two identical copies of the same program as independent threads either on a multithreaded processor or on different processors and compares their outputs for mismatch. Several researchers (e.g., [7],[6],[8],[9]) have proposed variations of this technique.

Thus, all current fault detection techniques use some form of redundancy that could have been used otherwise to boost the performance of the chip itself. If duplicate processor cores are used fault detection, then we sacrifice an entire processor's performance. If multithreaded processors are used, then we sacrifice about 20 – 40% of the processor's performance (e.g., [7],[8],[9]).

Fault recovery techniques can further reduce the performance of these processor chips. Fortunately, not every processor or environment would require fault recovery. Instead, on a detected fault, the processor could be halted and, then restarted. In such cases, fault recovery pays very little penalty in performance. (assuming fault recovery is not triggered too often). However, more aggressive and highly available systems may want to support aggressive fault recovery techniques, such as tri-modular redundancy (TMR) or pair-and-spare techniques. In TMR, outputs of three processors are compared for mismatch using a voting scheme. When a fault occurs in one of the processors, the other two processors' outputs would match and the program can continue executing. Of course, the processor that had a fault has to be restarted with the correct state, which may require synchronization with the other processors. In contrast, pair-and-spare uses two pairs of processors. Each pair

of processor has its own fault detection mechanism. When a fault is detected in one of the pairs, the other pair takes over as the main execution engine for the program. Thus, both TMR and pair-and-spare have very little recovery time. Other techniques, such as software-based recovery mechanisms, can incur significantly higher overhead in performance.

Clearly, the advent of CMPs with several processors on-chip can allow both fault detection and recovery techniques to be implemented on-chip. Fortunately, not all applications and environments will require aggressive fault detection and recovery mechanisms. Unfortunately, however, this could lead vendors to create several modes: with or without fault detection and with or without fault recovery. Once again, customers must be aware of what they are buying compared to which environment the processor has been benchmarked in.

4. POWER CONSTRAINT

Like cosmic ray strikes, increased power dissipation from microprocessor chips has also begun to plague the microprocessor industry. As the number of on-chip transistors and clock frequency increase, dynamic power dissipation from transistor switching begins to reach exorbitant levels. Borkar [11] predicts that dynamic power itself will increase from about 100W in 1999 to 2,000W in 2010. Worse, as supply voltage scales down, static power dissipation from leakage current starts becoming a major source of power dissipation as well.

Power dissipation has two implications—that of peak power dissipation and average power dissipation. The higher the peak power dissipation, the higher is the cost for packaging the chip. Additionally, higher peak power requirement may also dictate how many processor chips can be bundled in a certain cubic feet. The chips not only have to be cooled efficiently, but there must also be adequate power supplies to run these machines. Thus, densely-packed rack-mounted systems with several processors must closely monitor and reduce the power supply to and dissipation from these chips. Average power dissipation, on the other hand, has greater impact on the power supply, particularly if these machines are running on batteries.

Thus, to configure processor chips in certain packaging material and certain environments (e.g., densely-packed "blade" systems), vendors may create multiple modes with different power supply and dissipation characteristics for each mode. Designers have several tricks to reduce the peak power dissipation. For example, certain portions of the instruction queue, caches, and register ports could potentially be shut down in lower power modes. Also, vendors could reduce the frequency or reduce wasted work by choking the amount of speculation in processors with aggressive pipelines. In a CMP system, entire processors may be shut down to facilitate inclusion in lower power systems. Both architecture and circuit conferences abound in techniques to reduce the power supply to and dissipation from such chips.

Usually, however, the processor chips provide the highest performance in the highest power dissipation modes. Consequently, if vendors report the highest performance numbers from these chips, a customer may not be able to compare numbers for the same chips in lower power modes.

5. CONCLUSIONS

The above discussions suggest the advent of processor with several different hardware modes. Processor chips could support several orthogonal modes:

- Processors with snoopy protocol and processors with directory protocols,
- Processors with no fault detection, processors with fault detection but no transparent recovery, and processors with both fault detection and transparent hardware recovery, and
- Processors with high power mode and processors with low power mode.

The above combination itself would create 12 different hardware configurations. Unfortunately, only two of these configurations are the highest performing modes depending on the specific benchmark.

The above hardware modes can be classified into two categories: performance-centric and environment-centric modes. Performance-centric modes are those that provide best performance for specific sets of benchmarks. Thus, the choice between a snoopy and directory protocol falls under the performance-centric mode. The SPEC suite of benchmarks already allows vendors to provide two different software modes: *base* and *peak*. The base mode must use uniform compiler flags for all benchmarks, whereas the peak mode can use benchmark-specific flags. One solution to address the performance-centric hardware mode would be to allow vendors to include hardware flags in the peak mode. Then, customers can get some sense of the performance of the benchmarks they are interested in.

The environment-centric mode includes support for fault tolerance and support for different power modes. Customers would like to know the performance of

these processor chips in the fault tolerant modes and/or in low power modes. Then, customers must assess the environment they would operate these machines in and decide which processor chips to buy.

Currently, processor vendors are not required to report such numbers. Worse, a processor that has the highest performance in the absence of fault tolerance and in the high power mode may not be the highest performance processor when these features are turned on. Both vendors and customers must come together to decide what would be an appropriate benchmarking standard for this new generation of "system-on-a-chip" processors.

ACKNOWLEDGMENTS

I would like to thank Joel Emer and Geoff Lowney for their helpful comments on this paper.

REFERENCES

[1] IBM, "Power4 System Microarchitecture," http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html.

[2] David J.C.Johnson, "HP's Mako Processor," Fort Collins Microprocessor Lab, October 16, 2001. Available from http://cpus.hp.com/technical_references/mpf_2001.pdf.

[3] Shubhendu S. Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb, "The 21364 Network Architecture," Hot Interconnects IX, 2001.

[4] Alan Wood, "Data Integrity Concepts, Features, and Technology," White paper, Tandem Division, Compaq Computer Corporation.

[5] T.J.Slegel, et al., "IBM's S/390 G5 Microprocessor Design," IEEE Micro, pp 12–23, March/April, 1999.

[6] Todd M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," Proceedings of the 32nd Annual International Symposium on Microarchitecture, November 1999.

[7] Eric Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," Proceedings of Fault-Tolerant Computing Systems (FTCS), 1999.

[8] Steven K. Reinhardt and Shubhendu S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," International Symposium on Computer Architecture (ISCA), June-July, 2000.

[9] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," submitted for publication.

[10] Daniel P. Siewiorek and Robert S. Swarz**,** "Reliable Computer Systems: Design and Evaluation," A.K. Peters Ltd, October 1998.

[11] Shekhar Borkar, "Design Challenges of Technology Scaling," IEEE Micro, pp 23 – 29, July/August, 1999.

[12] Norbert Seifert, Compaq Computer Corporation, Personal Communication.

[13] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion, "Memory System Characterization of Commercial Workloads," Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA), pages 3 – 14, Barcelona, Spain, June/July 1998.

[14] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz Andre Barroso, "Performance of Database Workloads on Shared-Memory Systems with Out-ofOrder Processors," Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 307 – 318, San Jose, October 1998.

[15] Milo K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood, "Bandwidth Adaptive Snooping," Proceedings of the 8th Annual International Symposium on High-Performance Computer Architecture (HPCA), Feb 2002.

[16] J. Borkenhagen and S. Storino, "4th Generation 64-bit PowerPC-Compatible Commercial Processor Design," IBM Server Group Whitepaper, Jan 1999.

Session 2

Methodologies

Evaluating Non-deterministic Multi-threaded Commercial Workloads

Alaa R. Alameldeen, Pacia J. Harper, Milo M. K. Martin, Carl J. Mauer, Daniel J. Sorin, Min Xu, Mark D. Hill and David A. Wood
University of Wisconsin – Madison

How Input Data Sets Change Program Behaviour

Lieven Eeckhout, Hans Vandierendonck and Koen De Bosschere
Department of Electronics and Information Systems (ELIS)
Ghent University – Belgium

Benchmarking Web Server Architectures: A Simulation Study on Micro Performance

Haiyong Xie, Laxmi Bhuyan and Yeim-Kuan Chang
Department of Computer Science & Engineering
University of California - Riverside

Evaluating Non-deterministic Multi-threaded Commercial Workloads

Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M.K. Martin, Daniel J. Sorin,
Mark D. Hill and David A. Wood

Computer Sciences Department
University of Wisconsin — Madison
http://www.cs.wisc.edu/multifacet/

Abstract

Full-system simulation is increasingly used to evaluate the performance of commercial workloads on future multiprocessor designs. However, challenges such as simulation slowdown, sizing constraints, and workload tuning impede the development of commercial workloads for timing simulators. We describe how we address these challenges in our development of four commercial workload benchmarks.

This paper introduces non-deterministic workload behavior as another potential challenge in timing simulation. Non-determinism refers to the sensitivity of the system's timing to small changes in its parameters. This problem is nearly universally ignored because most simulators (including ours) are deterministic: they produce the same timing result every time, for given a workload and system parameters. However, we find that small changes in the memory latency can cause large changes in run-time (nearly 10%). We propose a methodology that uses pseudo-random perturbations and standard statistical techniques to compensate for these non-deterministic effects. Finally, we provide evidence that commercial workloads have different characteristics over time, further supporting a sampled simulation methodology.

1 Introduction

Commercial workload performance is an important metric for shared-memory multiprocessor computer systems. Full-system timing simulation is increasingly used to evaluate the performance of these workloads on future multiprocessor designs [11, 17]. However using commercial workloads in simulation environments requires addressing issues such as their long run times, large memory and disk requirements, and the complexity involved in tuning them.

Our workload development methodology (similar to prior work [5]) entails reducing simulation times, validating speed-ups, and scaling down the applications (as necessary). To reduce simulation time, we use simulation *checkpoints* to store a snapshot of the memory and disks after a long warm-up period. We use a *transaction-based* methodology that increases the accuracy of shorter simulation runs by reducing the start and end transient effects. To construct these workloads, we first tune them on an existing hardware platform, and then we load the *exact disk images* into our full-system simulation environment. These generic workload development concerns are addressed in Section 2, and the specifics of each workload is described in Section 3. We describe the target system and simulation infrastructure in Section 4, and present the properties of our workloads in Section 5.

The results in Sections 6 and 7 show that non-determinism exists in tuned multi-threaded commercial workloads, and that neglecting this behavior can result in incorrect conclusions. This problem is almost universally ignored because most simulators (including ours) are deterministic: they produce the same result every time, given the same workload and system parameters. Unfortunately, small changes in system parameters can expose the inherent non-determinism of the workload. For example, consider reducing the L2 cache miss latency of a *base* system by one cycle to produce an *enhanced* system. Intuitively, this enhancement should improve performance. However, this small change may result in completely different execution paths (e.g., due to lock races or external event timing), possibly resulting in worse performance. We demonstrate that this problem does indeed occur, then describe a methodology that uses pseudo-random perturbations and standard statistical measurement techniques to address this issue.

2 Workload Development

This section addresses the methodology we used to develop our commercial workload benchmarks. First, we describe how we shorten simulation time using checkpoints and a transaction-based measurement methodology. Second, we describe the two-step process of setting up and validating applications on an existing machine, then porting the applications into our simulation environment. Third, we describe our method for scaling down the workloads for tractable simulation.

2.1 Reducing Simulation Run Times

Running end-to-end simulations of commercial workloads would result in prohibitive run-times, due to the slowdown present in system-level multiprocessor simulation. For example, the TPC-C specification requires

that the benchmark runs for at least two hours on a real multiprocessor system [21]. Using a uniprocessor to simulate TPC-C on a 16-processor system would take more than 133 days, assuming a 1600x slowdown (100x per-processor). To limit simulation time, we use our simulator's checkpoint facility to capture the architectural state of the simulated system at the end of a warm-up period (which can take days to complete). All timing runs then start from the exact same checkpoint.

The commercial workloads presented in this paper are *throughput-oriented*. To measure throughput in a real system, one counts the number of transactions completed in a fixed time interval. For example, the TPC-C benchmark specification measures performance by the number of transactions completed per minute (tpmC) [21]. To measure throughput in our simulation environment, we instrument these workloads to signal the simulator at the end of each transaction using a special instruction. We then measure the amount of time it takes to complete a fixed number of transactions.

However, cold-start transients occur when transactions are executed before the system has reached a steady-state condition (e.g., the database buffer pool does not contain a frequently accessed index page). End transients occur when most processors become idle while they wait for the last several transactions to complete. To minimize these effects we warm-up the system, then take a checkpoint. Starting from this point, we measure the (simulated) time to complete a specified number of transactions. For example, benchmarking N transactions means time is measured and all processors continue execution until the Nth transaction completes. Even as this transaction completes, other transactions are in flight. By simulating a sufficiently large number of transactions, we reduce the effect of these partially complete transactions on the throughput. As we define transactions on a per benchmark basis, the work necessary to complete one varies between benchmarks. Two transactions in the same benchmark may represent significantly different amount of work (e.g., the "New Order" and the "Order Status" transactions in the OTLP benchmark).

2.2 Setup and Validation of Workloads

We set up and tuned our applications on a real multiprocessor machine, with the goal of developing workloads with reasonable multiprocessor speedup, and measuring microarchitectural statistics to be used in comparison with our simulation results. Using real machines makes setup much faster than under simulation, and permits rapid performance testing with different parameters. Due to the complexity of commercial workloads, they require a considerable amount of tuning in order to have reasonable speed-ups and scale-ups on multiprocessor systems and to avoid load imbalance.

We used a number of techniques to evaluate the applications' speed-ups and scale-ups, including hardware counters and operating system utilities. During this process, we used a Sun E6000 machine with sixteen 248-MHz UltraSparcII processors, each with a 1 MB unified L2 cache. UltraSparc processors have hardware performance counters that can be used to measure architectural events on a per-processor basis. We measured workload characteristics using these counters and calculated the Instruction Per Cycle (IPC) and cache miss rates as observed on the real machine. We tuned the workloads by seeking maximum speed-ups, measured by wall-clock run times for benchmark runs on a limited number of processors. For this purpose, we used the Solaris *psrset* tool to restrict given processes to run only on a subset of the available processors. Figure 1 shows the speedup of our set of workloads when running on 1, 2, 4, 8, and 16 processors. We also used the Solaris tool *mpstat* to measure the fraction of time spent by each processor in user, kernel, idle, or I/O wait mode. These utilization statistics allowed us to detect load-imbalance problems and verify that all processors are being sufficiently utilized (with less than 10% idle or I/O wait time). After this validation step was complete, we imported the exact disk images into our full-system simulation environment.

2.3 Scaling Down Workloads

Full-size commercial workloads can have memory requirements of many gigabytes, and secondary storage requirements of several terabytes. These memory and disk requirements often stress execution on native systems, and it is not currently possible to simulate such large systems. Moreover, simulators usually run on host workstations that are much less powerful than server systems. For example, our studies are conducted on PC-based Linux systems that have a 32-bit virtual address space limit.

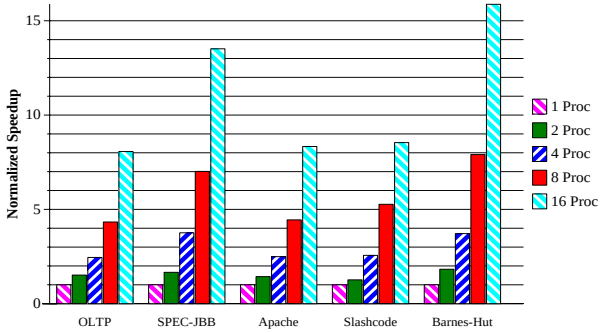


Figure 1. Workload parallel speedups

For these reasons, it is necessary to scale down the workloads so that they can be run in our simulation environment. We scaled down the workloads using trial and error to find the largest configuration of a workload that would run adequately in our simulator. Then we tested the performance of the scaled-down workloads on real hardware, to verify that the scaling has a minimal effect on workload behavior. For example, our OLTP benchmark (based on TPC-C) uses a 10-warehouse 1-GB database on five raw disks with a single log disk. Real TPC-C benchmark setups are normally two orders of magnitude larger in terms of the database size and number of disks used. Our setup on the Sun E6000 machine suffered a throughput penalty (in terms of tpmC numbers) of less than 30% compared to a 100-warehouse, 100 GB database constructed on 35 raw disks and 10 log disks. However, our simulation environment currently limits us to studying scaled-down versions of workloads.

3 Workloads

This section summarizes the set of workloads evaluated in this paper. This set includes two database on-line transaction processing applications, two web-server applications, and a scientific benchmark for comparison purposes.

3.1 OLTP

The TPC-C [22] benchmark models the database activity of a wholesale supplier, with many concurrent users performing business transactions against the database. The supplier operates out of a number of warehouses and their associated sales districts. The benchmark can be scaled by increasing the number of warehouses, but the database maintains fixed ratios of 10 sales districts per warehouse and 3000 customers per district. Transactions performed are of five transaction types, all related to the order-entry environment. Performance is measured by the number of "New Order" transactions performed per minute (tpmC), subject to certain constraints.

Our OLTP workload is based on the TPC-C v3.0 benchmark, but we scale down the data set. We use IBM's DB2 V7.2 EEE database management system and an IBM benchmark kit to build the database and model users. We build a 1 GB 10-warehouse database on five raw disks, and we use one additional dedicated disk for the database log. The TPC-C consistency requirements on the sizes of tables were maintained. We set the TPC-C client think time to be zero. We set the disk I/O latency in the simulator to be low and fixed (10 microseconds), emulating the high performance I/O subsystem in high-end servers.

We simulate 8 users per processor (e.g., 128 users on 16 processors), similar to Stets et al. [19]. Users are simulated using drivers from the IBM benchmark kit. A different process is started for each user. Each simulated user randomly executes transactions according to the TPC-C transaction mix specifications using a private random number generator. The database was warmed up by running for 10,000 transactions before taking measurements. Our results were based on runs of 1,000 transactions, unless otherwise specified. All completed transactions are measured, even those that do not satisfy some timing constraints in the original TPC-C benchmark specification.

3.2 SPECjbb

Java-based middleware applications are increasingly used in modern e-business infrastructure. SPECjbb is a Java program emulating a 3-tier system with emphasis on the middle tier. It fully implements the middle tier business logic. SPECjbb is inspired by the TPC-C benchmark and loosely follows the TPC-C specification for its schema, input generation, and transaction profile. SPECjbb runs in a single Java Virtual Machine (JVM) in which threads represent terminals in a warehouse. Each thread independently generates random input (tier 1 emulation) before calling transaction-specific business logic. The business logic operates on the data held in binary trees of java objects (tier 3 emulation). The specification states that the benchmark does no disk I/O or network I/O.

We used Sun's HotSpot 1.4.0 Server JVM and Solaris's native thread implementation. The system heap size was set to 1.8GB to avoid as much garbage collection as possible. Our experiments used 24 threads and 24 warehouses, with a data size of approximately 500 MB. The system was warmed up for 100,000 transactions, and our results are based on runs of 100,000 transactions.

3.3 Apache

Apache is a popular open-source web server used in many internet/intranet environments. In this benchmark, we focus on static web content serving.

We compiled Apache 1.3.19 on Solaris with GCC version 2.95.3. We made two compile-time changes to improve performance. First, we set Apache to use POSIX mutexes to serialize server processes waiting on `accept()`. Second, we set the dynamic module limit to zero, reducing the memory usage. We compiled and configured the Apache server according to the Apache group's performance notes [14].

We use the Scalable URL Request Generator (SURGE) [4] as the client. SURGE generates a sequence of static

URL requests which exhibit representative distributions for document popularity, document sizes, request sizes, temporal and spatial locality, and embedded document count. We ran 10 SURGE client threads per processor, and set the client think time to be zero.

Our experiments used a repository of 2000 files, with total size of approximately 50 MB, generated by SURGE using its default parameters. The system was warmed up for 80,000 transactions, and our results were based on runs of 2,500 transactions.

3.4 Slashcode

Dynamic web content serving has become increasingly important for web sites that serve large amount of information. Serving dynamic content is essential for online stores, instant news, and community message board systems. Our Slashcode benchmark is developed to represent these workloads.

Slashcode is an open-source dynamic web message posting system used by the popular slashdot.org message board system of the Linux user community. We used Slashcode 2.0, Apache 1.3.20, and Apache's mod_perl module 1.25 on the server side. MySQL 3.23.39 is used as the database engine. The server content is a snapshot from the slashcode.com site, and it contains approximately 3000 messages, with a total size of 5 MB. This benchmark is not database-oriented, so the size of the content has a small impact on system behavior. Most of the run time is spent on dynamic web page generation.

Autoslash is a multi-threaded user emulation program we developed to simulate user browsing and posting behavior. Each user independently and randomly generates browsing and posting requests to the server according to a transaction mix specification. There are 3 simulated users per processor. The system was warmed up for 240 transactions before taking measurements. Our results are based on runs of 50 transactions. Both server and client are compiled with Sun's WorkShop C 6.1 with aggressive optimization.

3.5 Barnes-Hut

For comparison, we selected one application from the SPLASH-2 [23] benchmark suite: Barnes-Hut with 64K bodies. The benchmark was compiled with Sun's WorkShop C 6.1 with profile-based feedback and uses the PARMACS shared-memory macros used by Artiaga et al. [3]. The macro library was modified to enable user-level synchronization through test-and-set locks rather than POSIX-thread library calls. We began measurement at the start of the parallel phase to avoid measuring initialization and thread forking.

4 Target System & Simulation Infrastructure

This section presents our simulation model of a target system and details of our simulation infrastructure.

4.1 Target System Model

We model a 16-node system similar to the Sun E10000 [7]. Each node contains a processor, caches, and an integrated memory controller for a portion of the 2 GB shared main memory. System caches are kept coherent using an MOSI invalidation-based snooping cache coherence protocol. We assume a single crossbar switch for the interconnection network to connect the nodes, with a delay of 50 ns for each interconnection network traversal (which includes wire propagation, synchronization, and routing). We selected 80 ns for memory DRAM access time. When a protocol request arrives at a processor or at memory, it takes 25 ns or 80 ns, respectively, to provide data to the interconnect. These assumed latencies result in a 180 ns latency to obtain a block from memory and a 125 ns latency for a cache-to-cache transfer. We assume an integrated processor and first level cache model that would complete four billion instructions per second if the memory system beyond the L1 caches was perfect. This establishes the relative speed of the memory system with respect to the processors, and is representative of a 2 GHz processor with a perfect L2-cache that has an IPC of 2. L2 caches are modeled as being 4 MB 4-way set associative with 64-byte blocks. We assume a 1 GHz system clock, hence the system cycle time is 1 ns.

4.2 Full-System Simulation

We use Simics [1], a full-system multiprocessor simulator, to simulate the same commercial workloads we set up on the real hardware. Simics is a system-level architectural simulator developed by Virtutech AB that is capable of booting unmodified commercial operating systems and running unmodified applications. We configured Simics to model an E6000-like SPARC V9 target system running unmodified Solaris 8. Simics is only a functional simulator by default, but it supports extensions to model timing. We use Simics' functional processor model to model a simple blocking processor that executes all instructions in one simulated processor cycle. We assume a processor clock that is four times the frequency of the system clock (i.e., 4 GHz). We extended this simple timing model with a memory hierarchy simulator that accurately models memory reference timing.

4.3 Memory System Simulator

Our memory system simulator, *Ruby*, processes requests from Simics and blocks memory operations on cache

Table 1. Workload properties

Workload	Memory blocks touched (64 bytes)	Unique miss PCs	L2 cache misses per 1000 instructions	Supervisor misses (% of total)	Time Spent in Kernel (% of total)
OLTP	57 MB	12136	3.0	43%	28%
SPECjbb	353 MB	8163	3.2	15%	1%
Apache	102 MB	10214	2.9	82%	84%
Slashcode	173 MB	17009	1.1	48%	43%
Barnes-Hut	16 MB	3413	0.3	16%	3%

misses. A blocked Simics processor will complete no instructions until Ruby explicitly un-blocks the processor when the miss processing completes. This allows Ruby to capture timing-dependent race conditions and lock contention that cannot be captured using a trace-driven methodology. Ruby supports a broad range of coherence protocols, which are specified using our table-driven specification methodology [18]. The specification is codified using our domain specific language *SLICC* (Specification Language for Implementing Cache Coherence) and software tools to generate the C++ source code for the protocol state machines used in Ruby. Using this methodology, our simulations capture timing races and state transitions (including transient states) of the coherence protocols in cache and memory controllers.

5 Workload Properties

Table 1 presents some properties of our workloads from simulations of a 16-processor system. This table corroborates previous results by showing that our OTLP benchmark spends approximately one-quarter of its time in the kernel [5], and that commercial applications have significantly worse cache performance characteristics compared to the Barnes-Hut scientific benchmark.

In the following sections, we present non-deterministic effects that we observed in the OLTP benchmark. The four commercial workloads studied in this paper exhibit these effects to different extents. Ranked from most variable to least variable, the commercial workloads are slashcode, OLTP, Apache, and SPECjbb. We chose to present OLTP's behavior as representative of these workloads.

6 Commercial Workload Non-determinism

When measuring real systems, researchers normally make multiple measurements and use standard statistical techniques to ensure statistically significant results [2]. The goal of this methodology is to ensure that uncontrolled experimental factors do not lead to false conclusions. For example, if the paging daemon runs in one execution but not in any others, using this methodology should isolate this non-systematic effect.

Conversely, when (most) researchers perform simulation studies, they (implicitly) assume that *all* experimental factors are controlled and present results from a *single* simulation run. This assumption seems plausible, since a deterministic simulator will always return the same result for a given combination of workload and system parameters. However, just because a simulator is deterministic does not mean that the workload is also deterministic. While a single-threaded, user-level application running on a uniprocessor may be deterministic, a multi-threaded commercial workload on a multiprocessor is not.

Workload non-determinism can be caused by small timing variations that cause the application or operating system to take different execution paths. On a uniprocessor, a small difference in the arrival time of a device interrupt may result in a radically different operating system scheduling decision. On a multiprocessor, synchronization races may cause the application or operating system to acquire locks in different orders. The outcome of memory races and scheduling decisions potentially leads to divergent executions, which may yield widely varying results for the simulated execution time after the completion of the same number of transactions, or may even execute different mixes of transactions. Since the amount of work required to complete a transaction in a given workload can vary, executing a different mix of transactions would likely result in different simulated execution times.

We performed an experiment that shows that small changes in memory latencies can affect process scheduling even in a deterministic uniprocessor simulation environment. We ran two OLTP simulations starting from the same checkpoint and artificially introduced instruction cache misses every 100 instructions. In the first run, these additional cache misses are introduced at instructions 0, 100, etc., while in the second they are introduced at instructions 50, 150, etc. While these two runs have the same cache miss rate, they report up to 9% dif-

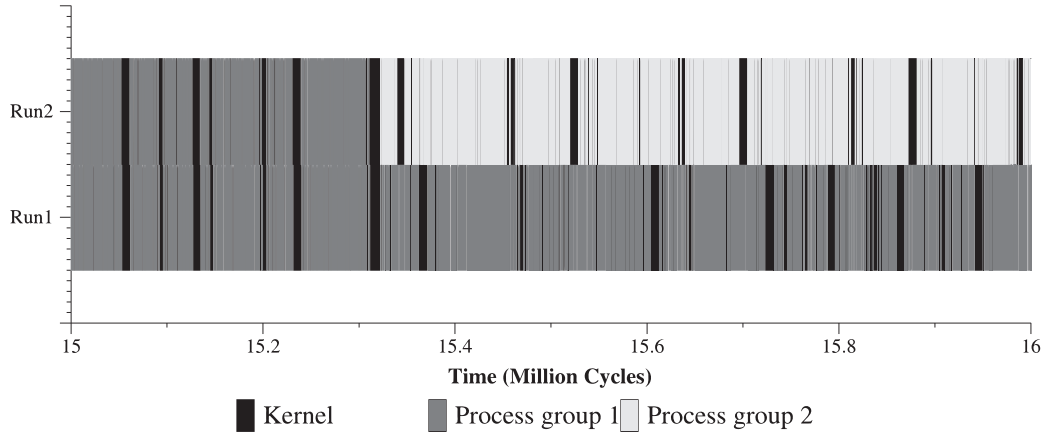


Figure 2. OS scheduling decisions are affected by memory latency

ference in simulated execution time after 2000 transactions. By instrumenting the simulator to report which processes are scheduled, we were able to show that the OS makes different scheduling decisions. Figure 2 shows a snapshot of the execution of these two runs. Run1 shows the execution of a process in process group 1, alternating between executing in kernel mode (black) or user mode (dark grey). In this same interval for Run2, the OS swaps out process group 1 and schedules a process of process group 2 (light grey). This snapshot shows the interval in which the initial divergence occurred. Both runs scheduled the same process groups prior to this snapshot, but the scheduling decisions were completely different beyond this point of divergence.

To mitigate the effects introduced by non-determinism, we run multiple simulations for each particular hardware configuration, and use their mean simulated execution time as our performance metric. We illustrate below how this methodology can be used to effectively separate systematic improvements from random effects caused by non-determinism.

Figure 3 presents statistics of the simulated execution time, gathered from a set of fifteen 1000-transaction OLTP runs on two different cache configurations. For each configuration, the left and right columns show the minimum and maximum (simulated) runtime, respectively. The center columns show the mean, with error bars indicating the standard deviation. Each run begins from the same checkpoint, but the simulator randomly introduces small perturbations in the memory system. On each L2 cache miss, the latency is randomly increased by a uniform random number between zero and seven nanoseconds. This perturbation effectively increases the contention-free memory access time to 183.5 ns and 128.5 ns, for memory and cache-to-cache misses respectively. Since the OLTP workload misses in the L2 cache no more than four times per 1000 instructions, the worst-case variation in CPI should be no more than 4%. And since there are millions of misses in each run, the law of large numbers [2] suggests that the actual variation should be much less. However, this perturbation results in widely varying execution times, whose range is approximately 10% of the mean. The minimum execution time for the direct-mapped case is lower (better) than the maximum for the set-associative case. If a researcher performed only a single simulation run for each case, s/he might draw the erroneous conclusion that the direct-mapped cache performs better than the set-associative cache. By performing multiple runs with random perturbations, statistically significant results can be obtained, that is the enhanced configuration performs better than the base case.

7 Workload Variability

Our transaction-oriented methodology simulates a given workload for a fixed number of transactions. A trade-off is made between simulation time and accuracy. Longer simulations amortize cold-start effects (e.g., cold

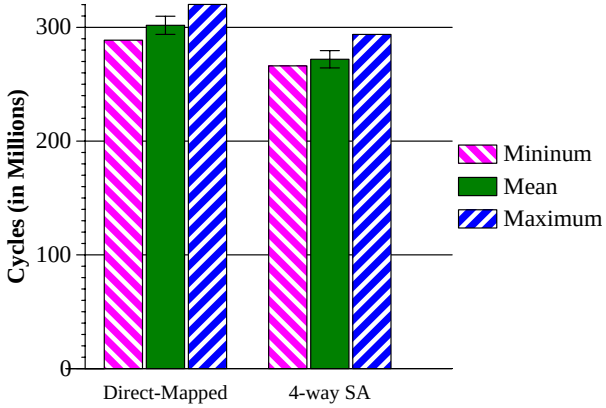


Figure 3. Execution time variations for two different cache configurations

Table 2. Some OLTP properties for different simulation lengths

Number of simulated transactions	200	400	600	800	1000	1200
System cycles per transaction	4.09	4.21	4.27	4.41	4.57	4.58
System cycles per instruction	0.78	0.74	0.72	0.70	0.66	0.66
L2 misses per 1000 instructions	3.79	3.50	3.39	3.23	2.99	2.98
L2 supervisor mode misses (%)	41.8	41.4	41.1	42.0	42.4	42.3
L2 misses per transaction (thousands)	19.78	19.99	20.11	20.48	20.70	20.73
Simulation runtime (hours)	1.92	3.88	5.89	8.02	10.18	12.23

caches) and smooth out variations due to heterogeneous transactions. To quantify this effect, we evaluated our workloads for different numbers of simulated transactions. Table 2 shows some architectural characteristics, computed from an average of twenty OLTP simulations on a 16-processor system. We had initially hoped that the runtime statistics would converge as we ran longer simulation. However, as shown in Figures 4, 5, and 6, even long simulations reveal that the workload exhibits different characteristics over time.

Using the OLTP workload, we simulated 8,000 transactions on a 16-processor system. These graphs show the average results of twenty separate runs, starting from the same checkpoint, measured every 200 transactions. The error bars indicate the standard deviations for each interval. Figure 4 shows there is variability during the run in the throughput of the system (the number of cycles to complete transactions). Figure 5 shows that the number of cache misses necessary to complete transactions is relatively stable across the run. Figure 6 shows there is considerable variability in the number of instructions executed to complete transactions in the OLTP benchmark.

Clearly a single short simulation run cannot capture the wide spectrum of the commercial workloads' behavior. Time-sampling is a well-known technique that may prove valuable to complete an architectural study within a reasonable simulation time [10, 12]. We intend to explore this further in future work.

8 Related Work

Prior work has studied commercial workloads for their architectural and micro-architectural characteristics, and has used them for simulation studies and for performance evaluations. The characterization studies can be classified by the level of detail in the processor model (in-order or out-of-order), their tools (real hardware or simulation), and the workloads studied. Our work distinguishes itself from these related works as it uses newer benchmarks, such as SPECjbb, larger configurations (in general), and studies these workloads' sensitivities to timing changes.

Barroso et al. [5] is an influential characterization study of commercial workloads on multiprocessors using both hardware counters, and in-order simulation tools. Ranganathan et al. [16] uses user-level out-of-order simulation to show that database workloads running on sequentially consistent systems can perform within 10-15% of release consistent systems. Keeton et al. [9] study the effects of out-of-order speculative execution on multiprocessor database workloads using hardware performance counters. Dedicated snooping hardware

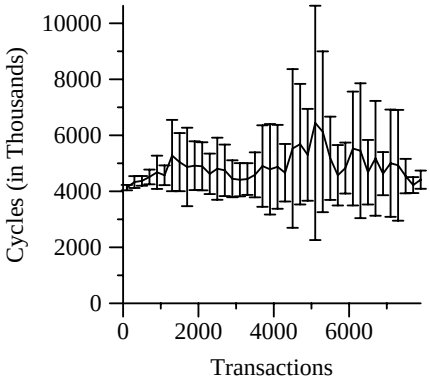


Figure 4. Cycles per transaction

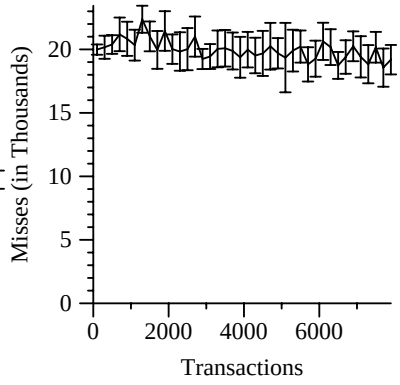


Figure 5. Misses per transaction

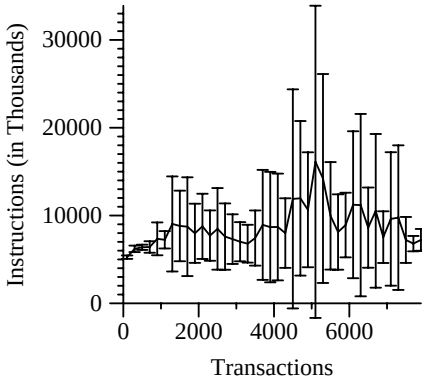


Figure 6. Instructions per transaction

has been used to study the memory system performance of commercial workloads running in real hardware systems [13]. The web-based on-line transaction processing benchmark TPC-W has also been studied in prior work [6]. The memory system performance of Decision Support System (DSS) workloads in multiprocessors have been characterized by Trancoso et al. [20].

In our simulation methodology we use an in-order processor model. Pai et al. [15] demonstrate that a key characteristic in shared-memory multiprocessor simulation is how the memory system is modelled. Durbhakula et al. [8] show this principle can be applied to approximate an out-of-order processor model using a simple processor model trading, achieving a significant speedup and introducing relatively little error.

9 Conclusions

This paper describes the methodology we adopt in developing four multiprocessor commercial workloads into benchmarks for simulation studies. In this methodology, we perform the setup of these workloads on real hardware for tuning and evaluation, then import them into our simulation environment. We provide a detailed explanation of our workloads and our simulation environment. We present some speed-up measurements of the tuned commercial workloads on the real hardware, and workload-specific tuning issues.

In order to deal with the non-determinism present in multiprocessor simulations, we proposed a transaction-based approach for simulation runs, that helps decrease cold-start and end transient effects. We demonstrate that even a fully deterministic simulation of a uniprocessor can exhibit non-deterministic behavior due to OS scheduling. We present how an incorrect conclusion can be reached if the issue of non-determinism is ignored. We demonstrate that commercial applications can exhibit different characteristics, and that these characteristics could lead to a large variation in simulation results. We propose a methodology for coping with this non-determinism by introducing minor perturbations in memory system latencies, and running multiple simulations for the same hardware configuration. This approach can lead to statistically significant results for these non-deterministic workloads.

Acknowledgements

This work is supported in part by the National Science Foundation with grants EIA-9971256, CDA-9623632, and CCR-0105721, an IBM Graduate Fellowship (Martin), an Intel Graduate Fellowship (Sorin), two Wisconsin Romnes Fellowships (Hill and Wood), and donations from Compaq Computer Corporation, IBM, Intel, and Sun Microsystems.

References

[1] Virtutech AB. Simics Full System Simulator. http://www.simics.com/.

[2] Arnold O. Allen. *Probability, Statistics, and Queueing Theory.* Academic Press, second edition, 1990.

[3] Ernest Artiaga, Nacho Navarro, Xavier Martorell, and Yolanda Becerra. Implementing PARMACS Macros for Shared Memory Multiprocessor Environments. Technical report, Polytechnic University of Catalunya, Department of Computer Architecture Technical Report UPC-DAC-1997-07, January 1997.

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

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

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

[7] Alan Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[8] Murthy Durbhakula, Vijay S. Pai, and Sarita V. Adve. Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 23–32, January 1999.

[9] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 15–26, June 1998.

[10] R. E. Kessler, Mark D. Hill, and David A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[11] Peter S. Magnusson et al. SimICS/sun4m: A Virtual Workstation. In *Proceedings of Usenix Annual Technical Conference*, June 1998.

[12] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 248–259, May 1993.

[13] Ashwini Nanda, Kwok-Ken Mak, Krishnan Sugavanam, Ramendra K. Sahoo, Vijayaraghavan Soundararajan, and T. Basil Smith. MemorIES: A Programmable, Real-Time Hardware Emulation Tool for Multiprocessor Server Design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[14] Apache Performance Notes. http://httpd.apache.org/docs/misc/perf-tuning.html.

[15] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 72–83, February 1997.

[16] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita Adve, and Luis Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307–318, October 1998.

[17] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, 1995.

[18] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, Anne E. Condon, Milo M.K. Martin, and David A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, To appear.

[19] Robert Stets, Luiz Andre Barroso, Kourosh Gharachorloo, and Ben Verghese. A Detailed Comparison of TPC-C versus TPC-B. In *Third Workshop on Computer Architecture Evaluation Using Commercial Workloads in conjunction with HPCA-6*, January 2000.

[20] Pedro Trancoso, Josep-L. Larriba Pey, Zheng Zhang, and Josep Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 250–260, February 1997.

[21] Transaction Processing Performance Council. TPC-C. http://www.tpc.org/tpcc/.

[22] Transaction Processing Performance Council. TPC Benchmark C, Draft Specification, Revision 4.0.q, August 1999.

[23] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

How Input Data Sets Change Program Behaviour

Lieven Eeckhout Hans Vandierendonck Koen De Bosschere
Department of Electronics and Information Systems (ELIS), Ghent University, Belgium
E-mail: {leeckhou,hvdieren,kdb}@elis.rug.ac.be

Abstract

Having a representative workload of the target domain of a microprocessor is extremely important throughout its design. The composition of a workload involves two issues: (i) which benchmarks to select and (ii) which input data sets to select per benchmark. Unfortunately, we are unable to select a huge number of benchmarks and respective input sets due to limitations on the available simulation time. In this paper, we use principal components analysis (PCA) to efficiently explore the workload space. Within this workload space, different input data sets for a given benchmark can be displayed and representative input data sets can be selected for the given benchmark. The final goal is to select a limited set of representative benchmark-input tuples that span the complete workload space.

1 Introduction

The first step when designing a new microprocessor is to compose a workload that should be representative for the set of applications that will be run on the microprocessor once it will be used in a commercial product [2, 12]. A workload then typically consists of a number of benchmarks with respective input data sets taken from various benchmarks suites, such as SPEC, TPC, MediaBench, etc. This workload will then be used during the various simulation runs to perform design space explorations. It is obvious that composing a representative workload is extremely important in order to obtain a design that is optimal for the target environment of operation. The question when composing a representative workload is thus twofold: (i) which benchmarks and (ii) which input data sets to select. In addition, we have to take into account that even high-level architectural simulations are extremely time-consuming. As such, the total simulation time should be limited as much as possible to limit the time-to-market. This implies that the total number of benchmarks and input data sets should be limited without compromising the final design. Ideally, we would like to have a limited set of benchmark-input tuples spanning the complete workload design space, which contains a variety of the most important types of program behaviour.

Conceptually, the complete workload design space can be viewed as a p-dimensional space with p the number of important program characteristics that affect performance, e.g., branch prediction accuracy, cache miss rates, instruction-level parallelism, etc. Obviously, p will be too large to display the workload design space understandably. In addition, correlation exists between these variables which reduces the ability to understand what program characteristics are fundamental to make the diversity in the workload space. In this paper, we reduce the p-dimensional workload space to a q-dimensional space with $q \ll p$ ($q = 2$ or $q = 3$ typically) making the visualisation of the workload space possible without losing important information. This is achieved by using principal components analysis (PCA). We will show that PCA can be used efficiently to explore the workload design space in general and to measure the impact of input data sets on program behaviour in particular.

Each benchmark-input tuple is a point in this q-dimensional space (obtained after PCA). We can expect that different benchmarks will be 'far away' from each other while different input data sets for a single benchmark will be clustered together. This representation gives us an excellent opportunity to measure the impact of input data sets on program behaviour. Weak clustering (for various inputs and a single benchmark) indicates that the input set has a large impact on program behaviour; strong clustering on the other hand, indicates a small impact.

In addition, this representation gives us an idea which input sets should be selected when composing a workload. Strong clustering suggests that a single or only a few input sets could be selected to be representative for the cluster. This will reduce the total simulation time significantly for two reasons: (i) the total number of benchmark-input tuples is reduced; and (ii) the benchmark-input tuple selected to represent a cluster can have a small dynamic instruction count compared to the other benchmark-input tuples in the cluster. The reduction of the total simulation time is an important issue for the evaluation using commercial workloads since commercial workloads tend to have large dynamic in-

struction counts in order to be representative.

This paper is organized as follows. In section 2, the program characteristics used are enumerated. Principal components analysis and its use in this context are discussed in section 3. In section 4, it is shown that PCA is useful in the context of workload characterization. In addition, we discuss how input data sets affect program behaviour. Section 5 discusses related work. We conclude in section 6.

2 Workload Characterization

It is important to select program characteristics that affect performance for performing PCA in the context of workload characterization. Selecting program characteristics that do not affect performance, such as the dynamic instruction count, might discriminate benchmark-input tuples on a characteristic that does not affect performance, yielding no information about the behaviour of the benchmark-input tuple when executed on a microprocessor. We have identified the following program characteristics:

- **Instruction mix.** We consider five instruction classes: integer arithmetic operations, logical operations, shift and byte manipulation operations, load/store operations and control operations.
- **Branch prediction accuracy.** We consider the branch prediction accuracy of three branch predictors: a bimodal branch predictor, a gshare branch predictor and a hybrid branch predictor. The bimodal branch predictor consists of an 8K-entry table containing 2-bit staturating counters which is indexed by the program counter of the branch. The gshare branch predictor is an 8K-entry table with 2-bit saturating counters indexed by the program counter xor-ed with the taken/not-taken branch history of 12 past branches. The hybrid branch predictor [18] combines the bimodal and the gshare predictor by choosing among them dynamically. This is done using a meta predictor that is indexed by the branch address and contains 8K 2-bit saturating counters.
- **Data cache miss rates.** Data cache miss rates were measured for five different cache configurations: an 8KB and a 16KB direct-mapped cache, a 32KB and a 64KB two-way set-associative cache and a 128KB four-way set-associative cache. The line size was set to 32 bytes.
- **Instruction cache miss rates.** Instruction cache miss rates were measured for the same cache configurations mentioned for the data cache.
- **Sequential flow breaks.** We have also measured the number of instructions between two sequential flow breaks or, in other words, the number of instructions between two taken branches.
- **Instruction-level parallelism.** To measure the amount of ILP in a program, we consider an infinite-resource machine, i.e., infinite number of functional units, perfect caches, perfect branch prediction, etc. In addition, we schedule instructions as soon as possible assuming unit execution instruction latency. The only dependencies considered between instructions are read-after-write (RAW) dependencies through registers as well as through memory. In other words, perfect register and memory renaming are assumed in these measurements.

For this study, there are $p = 20$ program characteristics on which PCA is performed.

3 Principal Components Analysis

Principal components analysisc (PCA) [17] is a statistical data analysis technique that presents a different view on the measured data. It builds on the assumption that many variables (in our case, program characteristics) are correlated and hence, they measure the same or similar properties of the program-input tuples. PCA computes new variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA tranforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the properties (i) $Var[Z_1] > Var[Z_2] > \ldots > Var[Z_p]$ which means that Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ which means that there is no information overlap between the principal components. Note that the total variance in the data remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[Z_i]$.

Because the total amount of variance is kept unchanged, some of the principal components will have only a small variance. Hence they have more or less the same value for all program-input tuples. By removing these components from the measurement data, we can reduce the number of workload characteristics, while controlling the amount of information that we throw away. We retain q principal components which is a significant information reduction since $q \ll p$ in most cases, typically $q = 2$ or $q = 3$. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $(\sum_{i=1}^{q} Var[Z_i])/(\sum_{i=1}^{p} Var[Z_i])$ accounted for by these q principal components.

During principal components analysis, one can either work with normalized or non-normalized data (the data is normalized when the mean of each variable is zero and

its variance is one). In the case of non-normalized data, a higher weight is given in the analysis to variables with a higher variance. In our experiments, we have used normalized data because of our heterogeneous data; e.g., the variance of the ILP is orders of magnitude larger than the variance of the data cache miss rates.

In this study the p original variables are the program characteristics mentioned in section 2. By examining the most important q principal components, which are linear combinations in the original program characteristics, meaningful interpretations can be given to these principal components in terms of the original program characteristics. To facilitate the interpretation of the principal components, we apply the *varimax* rotation [17] in the q-dimensional space. This rotation makes the coefficients a_{ij} either close to ± 1 or zero, such that the original variables either have a strong impact on a principal component or they have no impact. Note that the rotated principal components are still uncorrelated.

The next step in the analysis is to display the various benchmarks as points in the q-dimensional space built up by the q principal components. As such, a view can be given on the workload design space and the impact of input data sets on program behaviour can be displayed, as will be discussed in the next section.

We used STATISTICA'99 edition [1], a package for statistical computations, to perform PCA. This works as follows. A 2-dimensional matrix is presented as input to STATISTICA in which the columns represent the original variables, in our case the $p = 20$ program characteristics from section 2. The rows of this matrix represent the various program-input tuples, which will be enumerated in the next section. On this matrix, PCA is performed by STATISTICA which yields us the principal components. Once these principal components are obtained, it is up to the user to determine which principal components should be retained. This decision is made based on the amount of variance accounted for by the retained principal components. STATISTICA also computes the coefficients of the various program-input tuples as a function of the retained principal components. These data can be used to represent the program-input tuples in a q-dimensional space built up by these q retained principal components. The results of this analysis will be presented in the evaluation section of this paper.

4 Evaluation

4.1 Experimental Setup

In this study, we have used the SPECint95 benchmarks. The reason why we choose SPECint95 instead of SPECint2000 is to limit the simulation time since we wanted to work with reference inputs as much as possible. In addition to SPECint95, we used postgres v6.3 running the decision support TPC-D queries over a 100MB Btree-indexed database. For postgres, we ran all TPC-D queries except for query 1 because of memory constraints on our simulation machine. Details on the benchmarks and their input sets are given in Table 1.

The benchmarks were compiled with optimization level -O4 and linked statically with the -non_shared flag for the Alpha architecture. The characteristics measured in this paper are all dynamic characteristics. This was done by instrumenting these programs with ATOM [23], a binary instrumentation tool for the Alpha architecture.

4.2 Results

In this section, we will first perform PCA on the data for the various input sets of gcc. Subsequently, the same will be done for li and postgres, respectively. Finally, PCA will be applied on the data for all the benchmark-input tuples of Table 1.

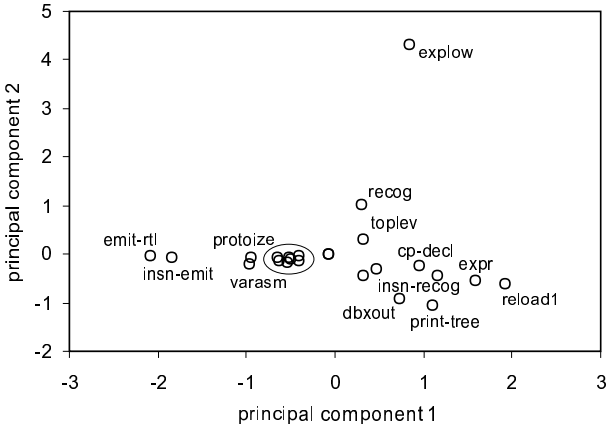


Figure 1. Gcc.

Gcc. PCA extracted two principal components from the data of gcc. These two principal components together account for 88.3% of the total variance. After varimax rotation, the first component is positively dominated, see Table 2, by the branch prediction accuracy, the percentage of arithmetic and logical operations; and negatively dominated by the I-cache miss rates. The second component is positively dominated by the D-cache miss rates, the percentage of shift and control operations; and negatively dominated by the ILP, the percentage of load/store operations and the number of instructions between two taken branches. Figure 1 presents the various input sets of gcc in the 2-dimensional space built up by these two components. Data

benchmark	input	dyn (M)	stat	mem (K)
gcc	amptjp	835	147,402	375
	c-decl-s	835	147,369	375
	cccp	886	145,727	371
	cp-decl	1,103	143,153	579
	dbxout	141	120,057	215
	emit-rtl	104	127,974	108
	explow	225	105,222	280
	expr	768	142,308	653
	gcc	141	129,852	125
	genoutput	74	117,818	104
	genrecog	100	124,362	133
	insn-emit	126	84,777	199
	insn-recog	409	105,434	357
	integrate	188	133,068	199
	jump	133	126,400	130
	print-tree	136	118,051	201
	protoize	298	137,636	159
	recog	227	123,958	161
	regclass	91	125,328	117
	reload1	778	146,076	542
	stmt-protoize	654	148,026	261
	stmt	356	138,910	250
	toplev	168	125,810	218
	varasm	166	139,847	168
postgres	Q2F	227	57,297	345
	Q3F	948	56,676	358
	Q4F	564	53,183	285
	Q5F	7,015	60,519	654
	Q6F	1,470	46,271	1,080
	Q7a	9.6	34,103	189
	Q8a	11.8	34,125	192
	Q9a	9.5	32,843	189
	Q10F	1,794	62,564	681
	Q11a	6.6	34,126	186
	Q12a	6.3	34,294	185
	Q13a	5.9	32,725	184
	Q14a	5.9	35,404	184
	Q15F	5.5	35,138	183
	Q16F	82,228	58,067	389
	Q17F	183	54,835	366
li	boyer	226	9,067	36
	browse	672	9,607	39
	ctak	583	8,106	18
	dderiv	777	9,200	16
	deriv	719	8,826	15
	destru2	2,541	9,182	16
	destrum2	2,555	9,182	16
	div2	2,514	8,546	19
	puzzle0	2	8,728	19
	tak2	6,892	8,079	16
	takr	1,125	8,070	36
	triang	3	9,008	15
ijpeg	vigo.ppm	817	16,037	1,273
	specmun.ppm	730	15,952	1,136
	penguin.ppm	790	16,128	1,227
go	50 9 2stone9.in	593	55,894	45
	50 21 9stone21.in	35,758	62,435	57
	50 21 5stone21.in	35,329	62,841	57
compress	5000000 e 2231	21,495	4,494	1,715
	1000000 e 2231	4,342	4,490	433
	500000 e 2231	2,182	4,496	272
	100000 e 2231	423	4,361	142
m88ksim	train.in	24,959	11,306	4,834
perl	jumble	2,945	21,343	5,951
vortex	train	3,244	78,766	1,266

Table 1. Characteristics of the benchmarks used with their inputs, dynamic instruction count (in million), static instruction count (number of instructions executed at least once) and data memory footprint in 64-bit words (in thousands).

points in this graph with a high value along the first component, have high branch prediction accuracies and high percentages of arithmetic and logical operations compared to the other data points; in addition, these data points also have low I-cache miss rates. Note that these data are normalized. Thus, only relative distances are important. For example, emit-rtl and insn-emit are relatively closer to each other than emit-rtl and cp-decl.

Figure 1 shows that gcc executing input set explow exhibits a different behaviour than the other input sets. This is due to its high D-cache miss rates, its high percentage of shift and control operations, and its low ILP, its low percentage of load/store operations and its low number of instructions between two taken branches. The input sets emit-rtl and insn-emit have a high I-cache miss rate, a low branch prediction accuracy and a low percentage of arithmetic and logical operations; for reload1 the opposite is true. The strong cluster in the middle of the graph contains the input sets gcc, genoutput, genrecog, jump, regclass, stmt and stmt-protoize. Note that although the characteristics mentioned in Table 1 (i.e., dynamic and static instruction count, and data memory footprint) are significantly different, these input sets result in quite similar program behaviour.

Li. PCA extracted two principal components from the data of the lisp interpreter. These two principal components together account for 75.6% of the total variance. After varimax rotation, the first component is positively dominated, see Table 2, by the percentage of memory operations, the

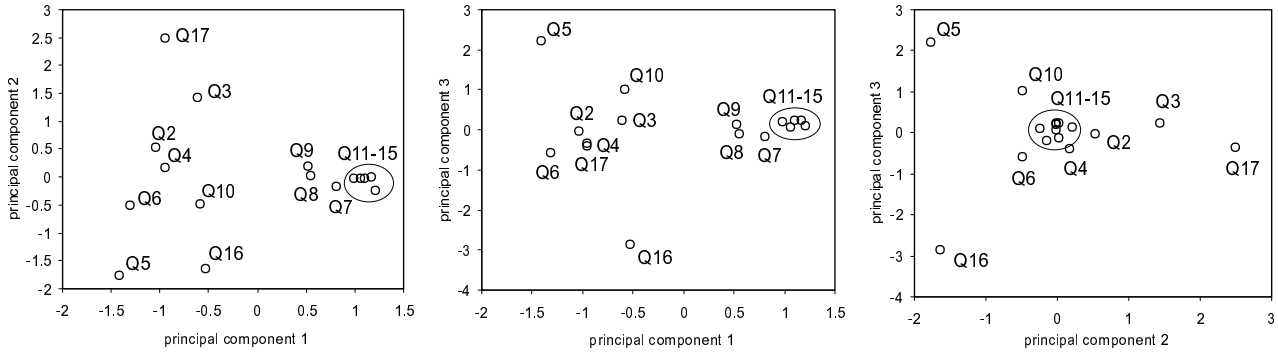


Figure 3. Postgres.

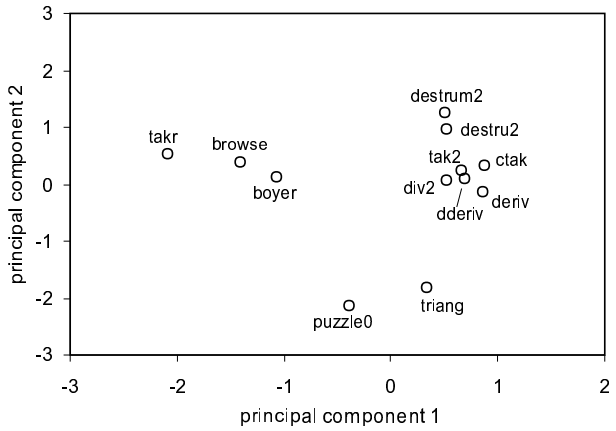


Figure 2. Lisp interpreter.

number of instructions between two taken branches and the miss rate for the 8KB I-cache; and negatively dominated by the percentage of logical and shift operations, and the miss rates for D-caches larger than 32KB. The second component is positively dominated by the amount of ILP and the miss rate for the D-caches smaller than 16KB; and negatively dominated by the percentage of arithmetic operations and the miss rate for I-caches larger than 32KB.

Figure 2 presents the various input sets of li in the 2-dimensional space built up by its two principal components. Five input sets result in a behaviour that is different from the other input sets located at the right of the graph. Three of these, namely takr, browse and boyer, have a higher miss rate for larger D-caches, a higher percentage of logical and shift operations, a lower percentage of load/store operations, a lower number of instructions between two taken branches and a lower miss rate for the 8KB I-cache. Two of these input sets, namely puzzle0 and triang, have a higher I-cache miss rate, a smaller miss rate for small D-caches, a higher percentage of arithmetic operations and a lower amount of ILP.

TPC-D. PCA extracted three principal components from the data of postgres running the TPC-D queries, accounting for 90.2% of the total variance. After varimax rotation, the first component is positively dominated, see Table 2, by the amount of ILP, the percentage of arithmetic operations and the D-cache miss rate; and negatively dominated by the branch prediction accuracy of the gshare branch predictor and the percentage of logical operations. The second component is postively dominated by the I-cache miss rate and negatively dominated by the percentage of shift and byte manipulation operations. The third component is positively dominated by the number of instructions between two taken branches and negatively dominated by the branch prediction accuracy and the percentage of control operations.

Figure 3 shows the data points of postgres running the TPC-D queries in the 3-dimensional space built by the three (rotated) components. In this graph, there is a strong cluster that contains queries 11, 12, 13, 14 and 15. From this graph, we can also conclude that queries 5, 16 and 17 exhibit a significantly different behaviour than the other queries. Query 17 has significantly higher I-cache miss rates and a significantly lower percentage of shift operations. Queries 5 and 16 differ from each other due to a high and low number of instructions between two taken branches, respectively; and a low versus high branch prediction accuracy and percentage of control operations, respectively.

Workload Space. Performing PCA on all benchmark-input tuples as described in Table 1, yields four principal components accounting for 89.2% of the total variance. After varimax rotation, the first component is positively dominated, see Table 2, by the branch prediction accuracy and the percentage of logical operations. The second principal component is positively dominated by the I-cache miss rates. The third component is positively dominated by the D-cache miss rates. The fourth component is positively

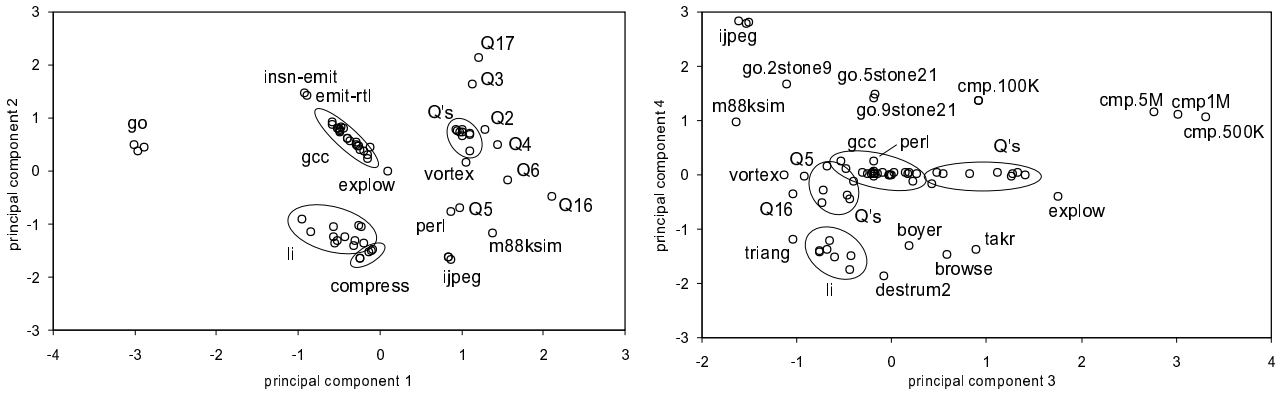


Figure 4. Workload space: first component vs. second component on the left and third vs. fourth component on the right.

dominated by the percentage of arithmetic operations and the number of instructions between two taken branches; and negatively dominated by the percentage of control operations.

Figure 4 shows these benchmark-input tuples in the four-dimensional workload space built up by its principal components. The benchmarks go, ijpeg and compress are somewhat isolated points in this 4-dimensional space. This is due to the high D-cache miss rates for compress, the high percentage of arithmetic operations, the low percentage of control operations and the high number of instructions between two taken branches for ijpeg and the low branch prediction accuracy and the low percentage of logical operations for go.

It is also interesting to note that the data points corresponding to the gcc benchmark are strongly clustered, except for the input sets emit-rtl, insn-emit and explow. This suggests that for the cluster gcc only a small number of input sets should be selected for the workload to represent the gcc benchmark, ultimately reducing the total simulation time of gcc compared to the simulation of all the input sets.

The data points corresponding to the lisp interpreter are strongly clustered as well, except for the following five input sets: triang, takr, destrum2, browse and boyer. The variety within these five input sets is caused by the data cache miss rate measured by the third principal component.

The data points corresponding to postgres running the TPC-D queries on the other hand, are weakly clustered except for queries 11 through 15 (not clearly seen on the graph). This suggests that for the TPC-D benchmark, several queries need to be considered in order to be representative. Note that all queries result in an above-average branch prediction accuracy (high value along the first principal component). The spread along the second principal component is very large and covers about 80% of the range of the second component. Therefore, a wide range of different I-cache behaviour can be observed when running the TPC-D queries. When inspecting the third principal component, we see that the TPC-D queries fall apart into two groups: those with relatively high data cache miss rates and those with relatively low D-cache miss rates.

The difference in behaviour between the program-input tuples for go and compress is mainly due to the difference in the data cache miss rates. For ijpeg, all three input sets seem to result in more or less the same behaviour.

In general, we can conclude that the variation between programs is larger than the variation between input sets for the same program. Thus, when composing a workload, it is more important to select different programs with a well chosen input set than to include various inputs for the same program. For example, the program-input tuples for gcc (except for explow) and ijpeg are strongly clustered in the workload space. In some cases however, for example for postgres running different TPC-D queries, the input set has a relatively high impact on program behaviour.

5 Related work

KleinOsowski *et al.*[14] propose to reduce the simulation time of the SPEC 2000 benchmark suite by using reduced input data sets. Instead of using the reference input data sets provided by SPEC, which result in unreasonably long simulation times, they propose to use smaller input data sets that accurately reflect the behaviour of the full reference input sets. For determining whether two input sets result in more or less the same behaviour, they used the chi-squared statistic based on the function-level execution profiles for each input set. Note that a resemblance of function-level execution profiles does not necessarily imply a resemblance of other program characteristics which are probably more di-

	li		gcc		TPC-D			all			
	PC1	PC2	PC1	PC2	PC1	PC2	PC3	PC1	PC2	PC3	PC4
ILP	0.11	**0.74**	0.43	**-0.70**	**0.87**	-0.13	0.33	-0.58	0.20	0.39	0.59
bimodal	0.38	-0.59	**0.94**	0.30	0.04	0.05	**-0.93**	**0.92**	0.20	-0.03	-0.18
gshare	0.44	0.09	**0.94**	0.22	**-0.70**	-0.10	-0.66	**0.78**	-0.35	0.01	-0.48
hybrid	0.67	-0.17	**0.95**	0.19	-0.45	0.07	**-0.83**	**0.80**	-0.07	-0.02	-0.56
ld/st	**0.91**	0.30	-0.58	**-0.80**	-0.59	0.14	0.66	-0.66	-0.34	-0.21	-0.58
int arithmetic	-0.25	**-0.96**	**0.94**	-0.06	**0.84**	0.48	0.10	-0.17	0.33	0.04	**0.81**
logical	**-0.84**	-0.41	**0.82**	0.12	**-0.90**	-0.39	-0.01	**0.95**	0.11	-0.08	0.01
shift	**-0.97**	0.17	-0.04	**0.86**	-0.04	**-0.72**	-0.47	0.52	-0.14	0.35	0.54
control	-0.23	0.64	0.24	**0.89**	-0.41	-0.32	**-0.80**	0.07	0.26	0.06	**-0.89**
seq. flow break	**0.82**	0.51	-0.14	**-0.95**	0.40	0.11	**0.89**	-0.26	-0.25	-0.12	**0.87**
I$ 8KB	**0.80**	-0.51	**-0.72**	-0.61	-0.04	**0.96**	-0.13	0.24	**0.88**	-0.13	-0.26
I$ 16KB	0.32	**-0.87**	**-0.76**	-0.55	-0.02	**0.98**	-0.13	0.28	**0.94**	-0.05	-0.01
I$ 32KB	0.04	**-0.86**	**-0.88**	-0.41	0.20	**0.93**	0.06	0.11	**0.97**	0.04	0.03
I$ 64KB	-0.05	**-0.92**	**-0.90**	-0.24	0.21	**0.86**	0.14	-0.08	**0.93**	0.00	0.05
I$ 128KB	-0.04	**-0.92**	**-0.85**	-0.14	0.52	**0.70**	0.25	-0.27	**0.75**	-0.04	0.06
D$ 8KB	0.09	0.63	0.27	**0.93**	0.67	0.66	0.27	-0.18	0.64	0.59	-0.08
D$ 16KB	-0.46	**0.75**	0.43	**0.88**	**0.83**	0.37	0.28	-0.14	0.46	**0.84**	0.01
D$ 32KB	**-0.93**	0.30	0.50	**0.86**	**0.95**	0.15	0.18	-0.04	-0.03	**0.99**	0.00
D$ 64KB	**-0.94**	0.27	0.57	**0.80**	**0.96**	0.06	0.13	0.04	-0.17	**0.97**	0.01
D$ 128KB	**-0.97**	0.17	0.63	**0.73**	**0.96**	0.00	0.10	0.12	-0.20	**0.95**	0.07

Table 2. The factor loadings of the principal components after varimax rotation for li, gcc, postgres and all the benchmark-input tuples, from left to right respectively. For example, this means that the first principal component for li is definied as follows: $PC1 = 0.11 \cdot ILP + 0.38 \cdot bimodal + 0.44 \cdot gshare + \ldots$ **Factor loadings greater than** 0.70 **are shown in bold.**

rectly related to performance, such as instruction mix, cache behaviour, etc. The latter approach was taken in this paper for exactly that reason. KleinOsowski *et al.* also recognized that this is a potential problem. The methodology presented in this paper can be used as well for selecting reduced input data sets. A reference input set and a resembling reduced input set will be situated close to each other in the q-dimensional space built up by the principal components.

Another important research topic that is related to this paper is trace sampling [5, 6, 13, 16]. In trace sampling, several samples are taken from a program execution so that the total number of instructions in the samples is significantly less than the total number of instructions of a complete execution. In order to make viable design decisions based on these sampled traces, a sampled trace should be representative for the complete program execution. Iyengar *et al.* [11] propose an R-metric for measuring the representativeness of a sampled trace. Lafage and Seznec [15] propose to choose representative samples using a data reduction technique, namely cluster analysis. The methodology presented here could also be used to validate sampled traces. Indeed, a sampled trace that is situated close to its reference trace in the workload space could be considered as being representative.

Only recently, a new fast simulation technique was introduced, namely statistical simulation [3, 7, 8, 19, 21, 20]. In statistical simulation, a statistical profile is extracted from a program execution which is subsequently fed into a synthetic trace generator. The synthetic trace being generated can then be executed on a trace-driven simulator which yields performance estimates. Due to the statistical nature of the technique, the total number of instructions in a synthetic trace can be limited since the performance characteristics while simulating a synthetic trace quickly converge. Typically, no more than one million instructions need to be simulated to obtain a stable performance estimate. Statistical simulation is related to the research topic presented in this paper, since the success of both techniques relies on choosing relevant program characteristics to be incorporated in the analysis. For statistical simulation, relevant program characteristics are needed to obtain a high accuracy; for the technique presented in this paper, relevant program characteristics are needed to construct a reliable workload space.

Another possible application of using a data reduction technique such as principal components analysis, is to com-

pare different workloads. In [4], Chow *et al.* used PCA to compare the branch behaviour of Java and non-Java workloads. The interesting aspect of using PCA in this context is that PCA is able to identify on which point two workloads differ.

Huang and Shen [10] evaluated the impact of input data sets on the bandwidth requirements of computer programs.

Changes in program behaviour due to different input data sets are also important for profile-guided compilation [22], where profiling information from a past run is used by the compiler to guide its optimisations. Fisher and Freudenberger [9] studied whether branch directions from previous runs of a program (using different input sets) are good predictors of the branch directions in future runs. Their study concludes that branches generally take the same directions in different runs of a program. However, they warn that some runs of a program exercise entirely different parts of the program. Hence, these runs cannot be used to make predictions about each other. By using the average branch direction over a number of runs, this problem can be avoided. Wall [24] studied several types of profiles such as basic block counts and the number of references to global variables. He measured the usefulness of a profile as the speedup obtained when that profile is used in a profile-guided compiler optimisation. Seemingly, the best results are obtained when the same input is used for profiling and measuring the speedup. This implies that every input is different in some sense and leads to different compiler optimisations.

6 Conclusion

In microprocessor design, it is important to have a representative workload to make correct design decisions. This paper proposes the use of principal components analysis to efficiently explore the workload space. In this workload space, benchmark-input tuples can be displayed. This representation can be used to measure the impact of input data sets on program behaviour. An interesting application for this technique is the selection of representative input data sets. Indeed, simulating a single or only a few representative input sets per benchmark instead of simulating a large number of input sets ultimately reduces the total simulation time.

Acknowledgements

Lieven Eeckhout and Hans Vandierendonck are supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT).

References

[1] StatSoft, Inc. (1999). STATISTICA for Windows. Computer program manual. `http://www.statsoft.com`.

[2] P. Bose and T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41–49, May 1998.

[3] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Workshop on Performance Analysis and its Impact on Design*, June 1998.

[4] K. Chow, A. Wright, and K. Lai. Characterization of Java workloads by principal components analysis and indirect branches. In *Proceedings of the Workshop on Workload Characterization (WWC-1998), held in conjunction with the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pages 11–19, Nov. 1998.

[5] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD-96)*, Oct. 1996.

[6] P. K. Dubey and R. Nair. Profile-driven sampled trace generation. Technical Report RC 20041, IBM Research Division, T. J. Watson Research Center, Apr. 1995.

[7] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 25–34, Sept. 2001.

[8] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, pages 1–6, Apr. 2000.

[9] J. Fisher and S. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, 1992.

[10] A. S. Huang and J. P. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 105–114, Oct. 1996.

[11] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 62–73, Feb. 1996.

[12] L. K. John, P. Vasudevan, and J. Sabarinathan. Workload characterization: Motivation, goals and methodology. In L. K. John and A. M. G. Maynard, editors, *Workload Characterization: Methodology and Case Studies*. IEEE Computer Society, 1999.

[13] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, June 1994.

[14] A. J. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held*

in conjunction with the International Conference on Computer Design (ICCD-2000), Sept. 2000.

[15] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held in conjunction with the International Conference on Computer Design (ICCD-2000)*, Sept. 2000.

[16] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical Report SMLI TR-93-22, Sun Microsystems Laboratories Inc., Dec. 1993.

[17] B. F. J. Manly. *Multivariate Statistical Methods: A primer*. Chapman & Hall, second edition, 1994.

[18] S. McFarling. Combining branch predictors. Technical Report WRL TN-36, Digital Western Research Laboratory, June 1993.

[19] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the third International Symposium on High-Performance Computer Architecture (HPCA-3)*, pages 298–309, Feb. 1997.

[20] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 15–24, Sept. 2001.

[21] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 71–82, June 2000.

[22] M. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proc. of ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 1–11, 2000.

[23] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, Mar. 1994.

[24] D. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the 1991 International Conference on Programming Language Design and Implementation (PLDI-1991)*, pages 59–70, 1991.

Benchmarking Web Server Architectures: A Simulation Study on Micro Performance

Haiyong Xie, Laxmi Bhuyan, and Yeim-Kuan Chang
Department of Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521
yong@cs.ucr.edu

Abstract

As Internet expands, the number of application servers, especially Web servers, has been increasing exponentially. To improve the performance of Web servers, researchers have paid attention to and studied the Web server's macro-performance, namely, the response time and throughput, which can be perceived by end users directly. In this paper, we have produced a micro benchmark, ServBench, by studying the micro performance of the most widely used Apache Web server. The bottleneck functions are identified by profiling the Apache server running with a realistic workload. We select some of these functions as micro-benchmark programs and study their characteristics. We port the microbenchmark to SimpleScalar simulation environment. We obtain execution time, branch prediction and cache miss results for the microbenchmark as a function of various architectural parameters.

1. Introduction

Recent years have seen an explosive growth of the Internet. Web applications and Web servers are critical to the success of the Internet. To improve the performance of Web servers, researchers have studied the Web server's macro-performance, namely, the response time and throughput, which can be perceived by end users directly. These studies have led to many benchmarks such as SPECweb [24], WebStone [26], NetPerf [20], and WebBench[27]. However, most of the macro-performance bottlenecks such as protocol stack overhead and process management overhead actually stems from the operating systems. Other studies show that web servers spend about 85% of the cycles in executing operating system codes compared to only 9% by SPEC95 suite [25]. Hu et al [14] found that Apache spends only 20-25% of the total CPU time on user code. This means Apache spends most of the CPU time in the kernel of operating system.

A number of performance evaluation studies on web servers have been reported in the literature. Most of these studies characterize external performance of web servers, namely, how the web server interacts with the outside world, which is called macro-performance in this paper. The workloads either consist of mainly static web page accesses or many static web page access blended with a small percentage of CGI scripts that perform very simple computation functions. A number of performance evaluation methodologies have been suggested in the literature [11,13,16,17].

The studies on improving Web server's macro-performance focus on improvement of either the interactivity between the Web servers, the underlying operating systems, or disk I/O and network I/O. The studies in this field generally fall into three categories: operating system enhancement [2,6,7,8], server program improvement [1,5], and caching techniques [12,15].

A very limited number of studies focus on architectural performance of Web servers, which we call micro performance in this paper. Radhakrishnan and John [23] evaluated the performance of Apache Web server in terms of micro-architecture using hardware performance monitoring counters. They studied such architectural performance as CPI and cache miss rates for both static and dynamic workloads. Iyer studied the cache performance of single and dual-processor servers running SPECWeb99 benchmark by feeding traces through simulation models of CacheFlowII [16]. The micro performance is very important for us to fully understand the impact of micro-architecture on the Web servers.

Another motivation, which is more important, is that we believe macro-performance improvement has its physical limitations like input/output processing and that we cannot exceed these limitations. Flash Web server [21] is claimed to be the fastest Web server and it outperforms existing Web servers by up to 50%. There is still much room to improve the Web server's micro performance

which can lead to better macro-performance. In order to understand how to improve the micro performance, we need to explicitly study the behavior of frequently used functions that contribute greatly to the execution time. A far out research will then be to develop assembly language instructions (like Intel MMX) for these functions or to build specialized hardware units on the CPU for fast execution of these functions.

To better understand the micro performance of web servers, we build an experimental environment for measuring the internal performance of a common modern Web server, Apache [2]. We take advantage of gprof [11], which is a program of GNU suite Unix tools [12], to get detailed profiling information of the Apache server. Through profiling the web server program, we are able to evaluate the performance of the web server in terms of its function calls. We identify the most time-consuming functions and the most frequently called functions. These functions are the bottlenecks to the server's micro performance and comprise the kernel of the web server.

Having known how much time these functions spend and how frequently they are called, we extract the top 8 function calls from the Apache server program and use them as the micro benchmark, ServBench. To make these programs run in a real system, we also extract their corresponding data structures together with the functions. All the benchmark programs need workload to operate on. We add some workload builder functions to the server program. When Apache serves incoming requests, the workload builder automatically generates the workload for benchmark programs based on the actual processing of the requests.

To know better the characteristics of the benchmark programs, we port and run the programs in the simulator, SimpleScalar [8]. As far as we know, ours is the first attempt to port a Web server benchmark program to an execution-driven simulator. By means of simulation, we obtain the characteristics such as instruction level parallelism, instruction frequencies, and cache performance for the micro-benchmark as a function of various architecture parameters. These characteristics are of great help to the design of high performance Web servers and Web-server-specific network processors.

We find that the average code size of ServBench is an order of magnitude smaller than that of SPECint. Both have similar instruction set characteristics. However, ServBench has smaller basic-block sizes and nearly half of the branches are taken and half not taken. This fact makes better branch predication mechanism and lower miss rate very important to the performance. We find that the Apache Web server can benefit tremendously from instruction level parallelism (ILP) because of the inherent parallelism of the ServBench programs. Also, L1 instruction cache plays a more important role than data cache in increasing the number of instructions executed per cycle (IPC). IPC is not sensitive to the set-associativity of instruction cache. We are able to achieve higher IPC by using asymmetric L1 cache, by enlarging the length of instruction fetch queue and by adding more ALUs. However, 4 ALUs and an instruction fetch queue of length 8 are enough to enhance the micro performance.

The rest of this paper is structured as follows. Section 2 describes profiling information of the measurement and how we achieve the ServBench. This section also gives a brief description of the experimental setup, how the web server services the requests and how we measure the architectural and functional performance using httperf [18] and GNU profile tools [12]. Section 3 presents the ServBench based on the profiling data in Section 2. Section 4 presents characteristics of benchmark programs including the instruction level parallelism, instruction frequencies, and cache performance. Section 5 concludes our work and suggests some future work.

2. Micro performance Measurement and Profiling

2.1 Experimental Setup

To measure and profile Apache Web server in the level of function calls, we have established an experimental environment which is comprised of a Linux server running Apache Web server, and several clients running the Web server benchmarking tool, httperf. The server and clients are connected to each other by a dedicated Ethernet using a 100Mbps Ethernet switch. This ensures that both the server and clients have access to enough network bandwidth available thus both have feasible high throughput and low response time. We have carefully chosen the benchmarking parameters for httperf including the request rate, number of requests, and number of connections so that the server reaches its possible highest throughput with the lowest load.

To get detailed profiling information, the Apache Web server is compiled by gcc 2.91 with function profiling option and optimization level O2. We use O2 level optimization for the reason that the compiler only performs target-processor independent optimizations and does not exploit particular architectural features such as loop unrolling for superscalar architectures.

The simulated processor architecture in SimpleScalar tool set is a close derivative of MIPS architecture. In all the simulations, the default issue width is 4; the default L1 caches are 4-way set-associative 16KB separate instruction cache and data cache; the line size of L1 cache is 32 bytes; the L2 cache is a 4-way set-associative 512KB unified cache with line size of 64 bytes; and the simulator is configured as out-of-order execution.

In addition to setting up the experimental network, server and clients, we also build the realistic workload for

the server according to SPECweb99 [24]. The workload consists of static files of four classes as shown in Table 1.

Classes	File Sizes	Target Mix
Class 0	less than 1K	35%
Class 1	less than 10K	50%
Class 2	less than 100K	14%
Class 3	less than 1000K	1%

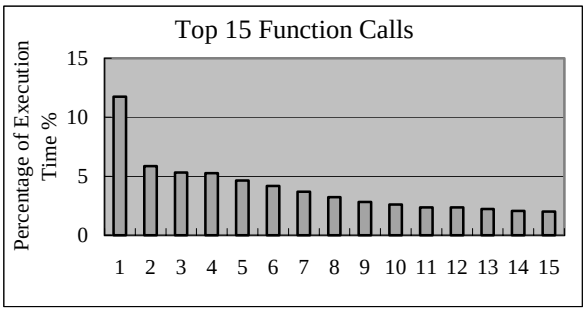
Table 1. File size mix of workload

We did not measure the performance with blended workloads which consist of both static and dynamic requests for a simple reason: our goal is to characterize the micro performance of the underlying processor and the internal performance of the server program. To study the micro and internal performance, we only need some simple but typical workloads which can be used to make the blended workloads. Dynamic requests always lead to the execution of some external programs such as Java or Perl CGI programs other than the Web server. The micro performance of those programs executed dynamically is not what we focus on.

2.2 Profiling Results

The clients in the experimental environment run httperf simultaneously to request a particular file class from the server. Apache Web server compiled with profiling options services the requests and writes the function profiling information to a specific binary output file. Later, the binary file can be converted to plain text file containing detailed profiling information using gprof. We collect all the function profiling information from the converted text file. We only pay attention to those functions that involve no disk I/O or network I/O activities directly since we focus on profiling the micro performance of Apache Web server.

After having collected profiling information for all the non-I/O functions, we rank the functions according to the percentage of execution time they account for. The top 15 most time-consuming functions account for almost 60% of execution time when the server services the HTTP requests, as shown in Figure 1.



	Function Name	%		Function Name	%
1	format_converter	12	9	check_hostalias	3
2	pstrdup	6	10	getword_white	3
3	run_method	5	11	process_item	2
4	config_log_transaction	5	12	conv_10	2
5	pstrcat	5	13	no2slash	2
6	palloc	4	14	getparents	2
7	table_get	4	15	get_module_config	2
8	invoke_handler	3			

Figure 1. Top 15 function calls

To our surprise, most of these functions are string processing related functions. Only a small part of these functions deal with the HTTP requests directly. This is feasible because HTTP protocol is a text-based protocol and the processing of HTTP protocol is essentially processing of strings. Some of these functions are sub-functions of others, for instance, process_item and conv_10 are sub-functions of config_log_transaction and format_converter, respectively.

3. ServBench

3.1 Selection of Benchmark Programs

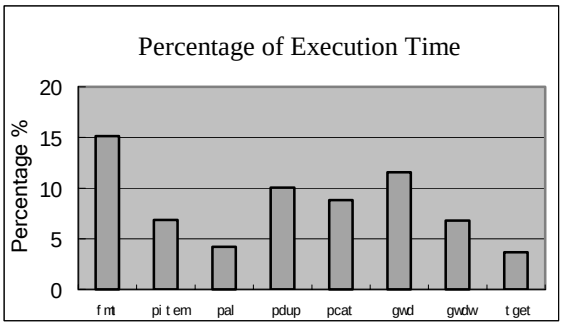
Most of the functions in Figure 1 have supporting sub-functions, for example, format_converter always calls conv_10 to convert integer numbers and conv_fp to convert floating-point numbers. Thus, we create fmt by combining the main function, format_converter, with such supporting functions as conv_10.

Some of the top 15 functions are just interfaces to a group of functions, for example, run_method and invoke_handler are called to invoke other functions indirectly using function pointers, which can not be identified by gprof. We do not consider these functions because they spend very little time in invoking other functions.

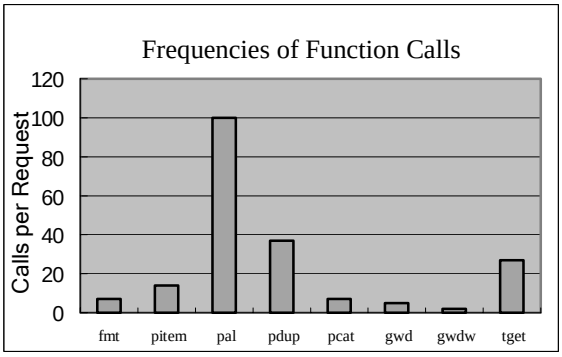
After having combined sub-functions with main functions and deleted interface functions such as run_method, we obtain eight sets of programs that are most time-consuming, fmt, pitem, pal, pdup, pcat, gwd, gwdw, and tget, as shown in Figure 2.1. These programs are ranked according to the percentage of execution time they spend. They account for 40% of total execution time (without considering disk and network I/O time). In this paper, we take these sets of programs as single functions for clarity.

These eight programs have different frequencies as shown in Figure 2.2. String allocation (pal) and duplication (pdup) functions are called 100 and 36 times respectively during Apache processes an incoming request. However, other functions such as fmt account for more execution time compared to pal although they are called less frequently. The reason is that fmt is about an order of

magnitude larger than pal and pdup in the size of source code and runtime kernel, as we will see in Section 5.2.



(1) Percentage of execution time



(2) Call frequencies

Figure 2. Benchmark programs: percentage of execution time and call frequencies

We build the micro-benchmark, ServBench, based on the above eight programs. There are several reasons for which we choose them as the benchmark programs. First, They are the most time-consuming elements of Apache server. If we want to improve the micro performance of Apache server, we will have to improve the performance of these functions because they are the bottleneck functions. Secondly, they are very frequently called when Apache processes the requests. Thirdly, benchmark programs should represent a wider application class in the domain of interest. The above functions are general functions to process the HTTP requests. Since HTTP protocol is basically a text-based protocol, all the request lines and header lines in the protocol payload are texts; to generate the response headers and log the requests are text-based as well. We believe every implementation of Web servers needs to process the text-based request lines and header lines very frequently. These string processing functions are representatives of HTTP protocol processing applications.

These programs can be divided into three groups. Group 1 has fmt and pitem as string format conversion programs; group 2 has pal, pdup, and pcat as string generation programs; group 3 consists of gwd, gwdw, and tget as string comparison programs.

3.2 String Format Conversion

String format conversion functions represent the operations of converting a number of values to a string. These functions are called to generate a response header or to log the corresponding request. String format conversion functions include fmt and pitem. Fmt is used to convert all other types of data to a string, e.g., converting an integer to its corresponding printable string, converting the request time to a string, and generating weak Etags for response headers. Pitem is used to generate the log entries for each of the incoming requests, for example, request line, request time, client address, and status of responses, etc.

3.3 String Generation

String generation functions, which include allocating a memory block for a new string, duplicating a string, and concatenating a number of strings, are very frequently called during the process of incoming HTTP requests. As mentioned before, HTTP protocol payload is text-based strings. To manipulate the payload, the content of payload which is comprised of many strings, has to be duplicated and stored in user space buffers. For instance, Web servers have to keep the state of a request in memory which may consist of the request string and some of the header strings. Web servers also need to log the requests which requires keeping some of the request strings in memory. These functions are also used in generating response headers.

String generation functions include pal, a string allocation function, pdup, a string duplication function, and pcat, a string concatenation function.

3.4 String Comparison

String comparison functions are very often called as well. These functions are used to extract a part from a long string which consists of many sub-strings separated by delimit characters such as blank space or colon. Tget, gwd, and gwdw are the three programs in this category. Gwd and gwdw are called to extract a “word” from a string which comprises many words separated by blank spaces or other predefined delimit characters. Tget is used to retrieve the corresponding value of headers from the HTTP requests. Since each request may have many headers, which again consists of may key strings and their corresponding value strings, tget is called to get the value string for a specific key string.

4. Benchmark Characteristics

We have selected the following general areas of characterization for further consideration: program code

sizes and kernel sizes, instruction set characteristics, instruction level parallelism, and cache performance.

4.1 Methodology

We use SimpleScalar to study the characteristics of these benchmark programs. SimpleScalar is an execution-driven simulator package commonly used by computer architecture researchers. SimpleScalar has its own C compiler (a modified version of GNU GCC) with associated utilities. We run the programs in the simulator and get the detailed performance data by changing the architectural parameters such as cache size, number of ALUs, etc.

All the programs have to have some data to deal with. To generate datasets for them, we insert some small functions into Apache server to gather all the data needed by each benchmark program. By doing so we are able to get the data that is dealt with by Apache derived from the realistic workloads, which is meaningful to characterizing the benchmark. This method has a potential valuable property: we are able to update the associated data for each benchmark programs very easily as the workload changes.

4.2 Program Kernel Size

Knowing the sizes of program kernels is useful for us to learn the static properties such as the number of lines of C code and size of compiled code, and dynamic properties such as instructions executed at least once and instructions accounting for 99% of execution time. We compare ServBench programs to SPECint programs as well.

Table 2 shows the size of the C source code and compiled executable of each benchmark program in both ServBench and SPECint. The object code size does not include dynamically linked libraries.

The average code size of ServBench programs is 28,679 bytes which is nearly an order of magnitude smaller than that of SPECint programs. The differences in code sizes of ServBench and SPECint programs come from the different environments where the applications or functions have been implemented and executed. String generation functions such as pal, pdup, and pcat are the most frequently referenced functions in Apache; they have rather simple functionalities compared to other function calls and applications. Other programs such as string format conversion and comparison functions are larger in code sizes in average. However, the SPECint programs are actual applications in real systems. They all have much more complex functionalities thus have much larger object code sizes.

ServBench	Code Size (C Lines)	Code Size		SPECint	Code Size (C Lines)	Code Size
Fmt	3,206	76732		126.gcc	206,000	1950000
Pitem	1396	20040		130.li	7,600	139000
Pal	326	9644		099.go	29,200	558000
Pdup	320	9628		134.perl	26,900	544000
Pcat	327	9992		124.m88ksim	19,900	404000
Gwd	427	75876		147.vortex	67,200	1150000
Gwdw	424	19560		132.ijpeg	31,200	594000
Tget	174	7960		129.compress	19,300	81700
Average	825	28679		Average	48700	678000

Table 2. Code sizes of ServBench and SPECint

ServBench	Instructions at Least once	Instructions For 99%		SPECint	Instructions at Least once	Instructions For 99%
Fmt	18400	1112		126.gcc	124246	15899
Pitem	1767	206		130.li	7341	408
Pal	660	198		099.go	12627	949
Pdup	657	226		134.perl	12313	875
Pcat	713	269		124.m88ksim	12284	542
Gwd	992	293		147.vortex	60630	1715
Gwdw	990	324		132.ijpeg	53629	6530
Tget	384	57		129.compress	2842	227
Average	3070	335		Average	35700	3390

Table 3. Dynamical Properties of ServBench and SPECint programs

The dynamical kernel size of ServBench is an order of magnitude smaller than SPECint as well, as shown in Table 3. A common rule, "90/10 rule", can be seen from the table in average: 90% of executed instructions are derived from 10% of the instructions in the program. Most of the programs have a relatively small kernel that account for most of the execution time.

4.3 Instruction Set Characteristics

The instruction mix gives indications on the types of instructions executed in the benchmark. Figure 3 presents the frequencies of different types of instructions for each ServBench program (instruction types are noted in the figure). Averages for each of the three groups of benchmarks, ServBench, and SPECint are also given in Figure 4.

These two benchmarks have similar instruction set characteristics in terms of the general trend and variability. The average difference in frequencies between ServBench and SPECint is between 5% (13% store instructions in ServBench while 9% in SPECint) and 1% (42% integer computation instructions in ServBench versus 43% in SPECint).

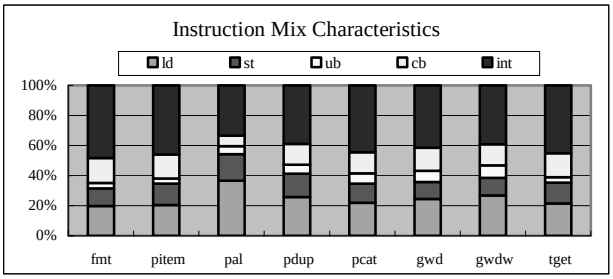


Figure 3. Instruction mix characteristics (ld=load, st=store, ub=unconditional branch, cb=conditional branch, int=integer computation. No floating-point instructions)

There are significant differences between the three groups of benchmarks, Apache, and SPECint that can be seen in from Figure 4.

The three groups of sub-benchmarks have different instruction execution frequencies. For instance, G1 has 8% percent less load instruction than G2 and 4% less than G3, however, it has 8% percent more integer computation instructions than G2 and 5% more than G3. But all these three groups have very similar percentage of store instructions. Among the three groups, string comparison functions have similar trend and variance to SPECint. The difference is under 3% (12% store instructions in group 3 compared to 9% in SPECint). Other groups have much significant and variable differences ranging from 1% to 6% in each instruction type.

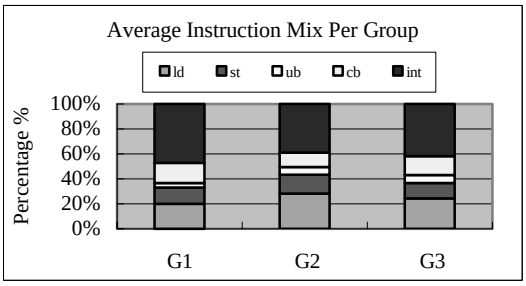


Figure 4. Average instruction mix (G1=string format conversion, G2=string generation, G3=string comparison)

4.4 Instruction Set Characteristics

Instruction level parallelism (ILP) is an important issue in improving a Web server's micro performance. Knowing the benchmarks' instruction level parallelism can be of great help to the design of application specific processors and architectures such as network processors.

We obtain the instruction level parallelism for each benchmark program by changing such parameters as the length of instruction fetch queue, the number of ALUs, and branch prediction mechanism. All these parameters have important impact.

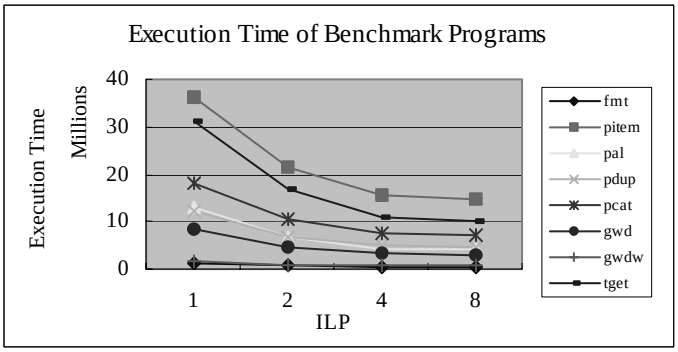


Figure 5. Execution time of benchmark programs

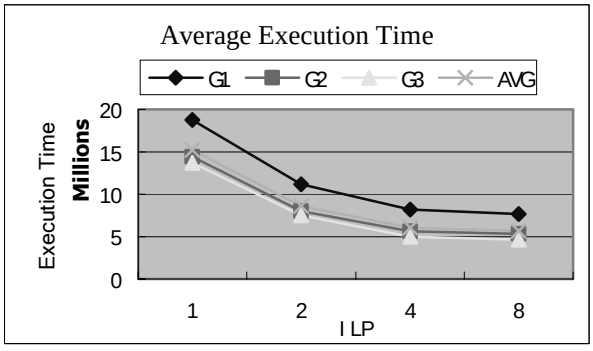


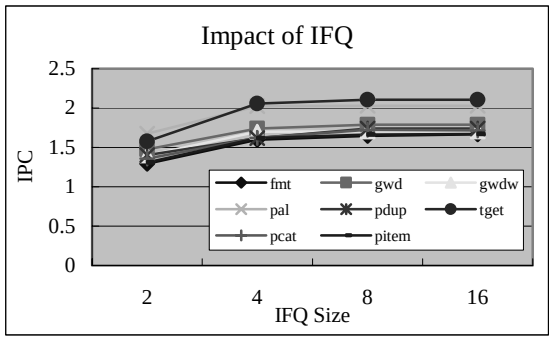
Figure 6. Avg. execution time of benchmark programs

Figure 5 shows the relationship between execution time and ILP for each benchmark program. Figure 6 depicts the average execution time of each group of functions. It is observed that it is almost enough for us to achieve the best performance when ILP is 4.

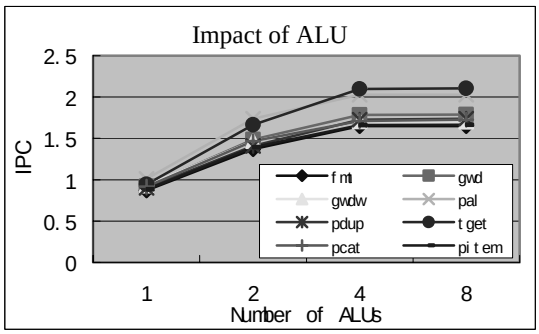
Figure 7.1 and 7.2 depict the impact of instruction fetch queue and ALU respectively.

With 8 ALUs and decode/issue bandwidth of 8 instructions per cycle, the highest instruction per cycle (IPC), which is 2.1, can be reached when instruction fetch queue is 8. Increasing the length of the queue is of no help to enhance the performance. An instruction fetch queue of length 8 is enough to achieve best performance in this case. On the other hand, 4 ALUs are enough for achieving best performance if the instruction fetch queue is 16 and decode/issue bandwidth is 8. Increasing the number of ALU does not help to improve the performance in terms of IPC.

From the above observations, we see that 4 ALUs and instruction fetch queue of length 8, or an ILP of 4, are enough for best performance. However, there are intrinsic reasons for this.



(1) Impact of instruction fetch queue



(2) Impact of ALU

Figure 7. Impact of instruction fetch queue and ALU

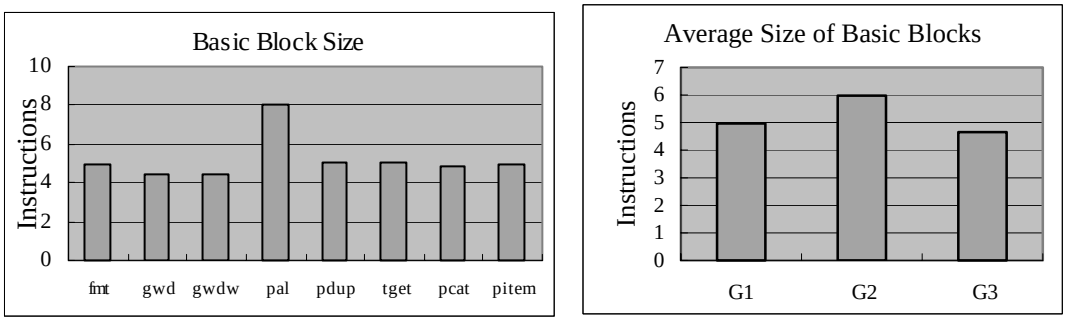
Figure 8 shows the size of basic blocks of each benchmark program and group. Most of the programs' basic block sizes are less than 5 except pal which has the size of 8. The three groups of programs have average size of 5, 6, and 4.6 respectively. This means every 5 or 6 instructions in the instruction queue must have a branch which is taken with a probability of nearly 50%, as shown in Figure 9. Thus an instruction queue of length 8 has an effective length of 4 due to half of the branches are taken and the other half not taken. 4 ALUs are enough for best performance for two reasons. One reason is that the effective length of the instruction queue is only 4 which means there are at most 4 instructions decoded and issued to the ALUs. The other reason is that only 40% instructions are integer computation instructions and that the maximum decode/issue bandwidth is 8 instructions per cycle, which means less than 4 instructions per cycle are in need of ALU operations.

The miss rate of branch prediction mechanism has much greater impact on IPC than expected. Both predict-not-taken and predict-taken have nearly the same high miss rate as shown in Figure 9.1. The bimod, 2lev and combined techniques predict with an accuracy between 80% to 100% for different benchmarks. From Figure 9.2 we can see that IPC reaches more than 2 for most of the programs with perfect branch prediction mechanism. However, even with the best branch-prediction mechanism, combining bimod with 2lev, IPC can only reach 86% of IPC with perfect branch prediction. There is much room for improving micro performance by means of improving branch prediction hit rate.

4.5 Instruction Set Characteristics

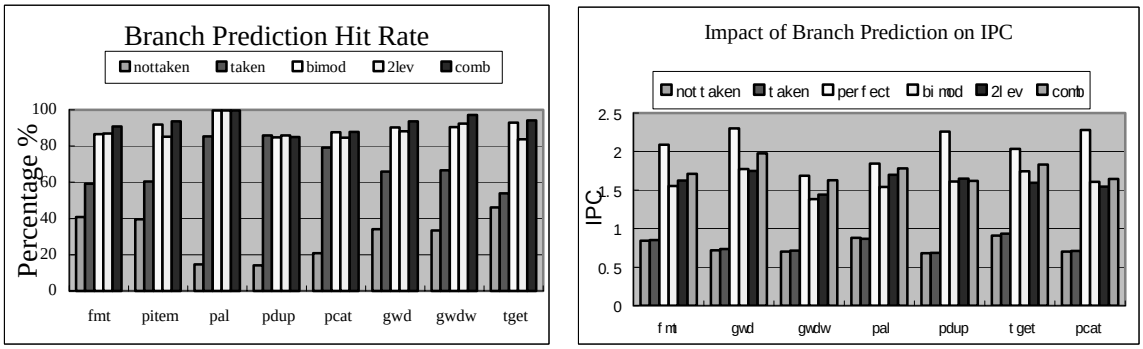
Cache behavior is very important to the micro performance. We measured the cache performance for each ServBench program. Separate L1 instruction cache and data cache were simulated. The size of data cache ranges from 2KB to 256KB, that of instruction cache ranges from 2KB to 64KB in Figure 10 and Figure 11, which show the miss rates for a 4-way associative data cache and instruction cache respectively. The results are shown in terms of groups.

It seems that the size of instruction cache has greater impact than data cache. The instruction miss rates for small caches sizes are much higher than the corresponding data cache miss rates. When cache size increases from 16KB to 32KB, data cache miss rate decreases 43% in average (26% for G1, 34% for G2, and 58% for G3), however, instruction cache miss rate decreases 76% in average (52% for G1, 25% for G2, and 98% for G3). When cache size increases from 32KB to 64KB, miss rates of data cache and instruction cache decrease 23% and 63% respectively in average. Large instruction cache favors ServBench.



(1) Basic-block size per program (2) Basic-block size per group

Figure 8. Size of Basic Block of ServBench programs



(1) Branch-prediction Hit Rate (2) Impact of Branch Prediction on IPC

Figure 9. Impact of Branch Prediction

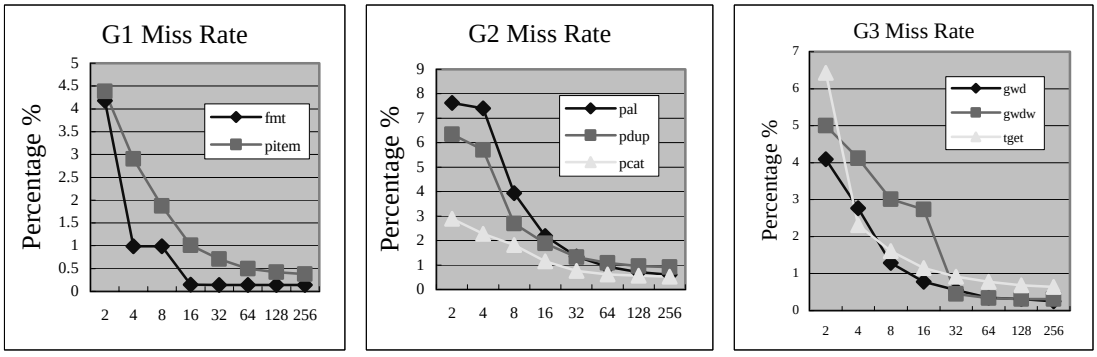


Figure 10. L1 Data Cache Miss Rate as a function of cache size

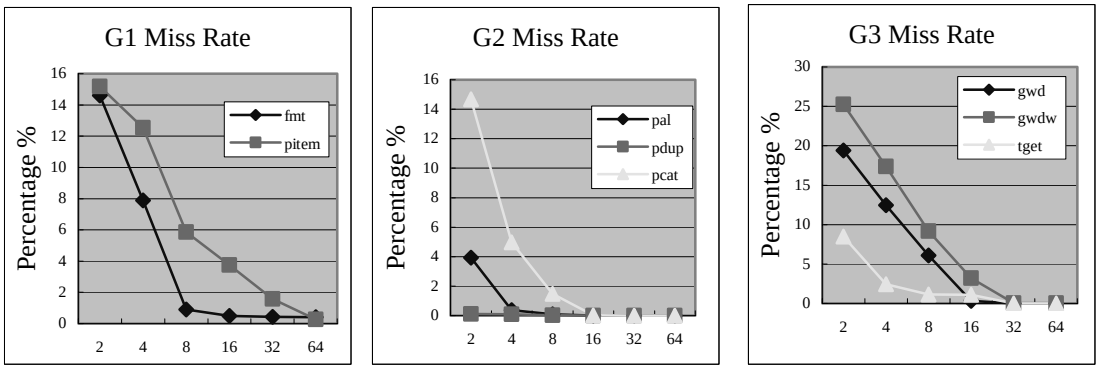


Figure 11. L1 Instruction Cache Miss Rate as a function of cache size

Figure 12 compares the ServBench, SPECint, and the three groups of benchmark programs. We need at least 16KB instruction cache to obtain the miss rate under 2% and 32KB to lower the miss rate to 1%. Although the kernels of ServBench programs are small, there are a lot of standard library functions called by the kernels which makes the instruction cache miss rate higher than expected. Compared to SPECint, only G2 programs, which have the smallest kernel, have lower miss rate.

Compared to instruction cache behavior, data cache performance of ServBench is more similar to that of SPECint. The data cache miss rates for ServBench are roughly half that of SPECint.

4.6 Instruction Set Characteristics

It seems that L1 instruction cache has greater impact on the micro performance. From cache performance results we observe larger instruction cache favors ServBench. We obtain memory access behavior measured by memory accesses per instruction (MAPI), as shown in Figure 13.

Figure 13 shows the memory access behaviors of benchmarks and Apache. We define memory access per instruction (MAPI) as the ratio of number of memory references for data to the number instructions executed in a run. MAPI represents the frequency of memory accesses in terms of instruction execution. ServBench has a similar MAPI as SPECint with a variance of less than 3%.

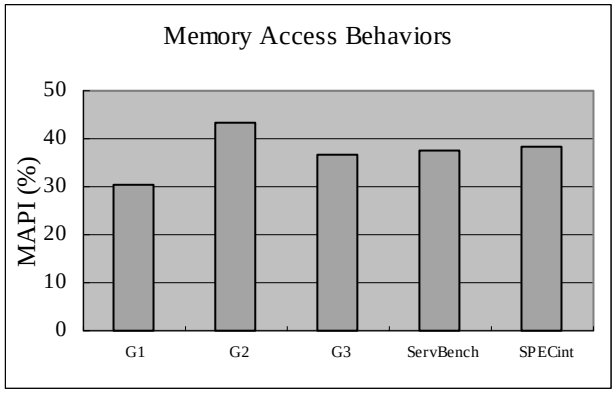


Figure 13. Memory Access Behaviors

Based on the above results and observations, it is possible to use asymmetric L1 caches to improve the micro performance. Most of present processors have symmetric L1 caches. For example, the mainstream microprocessor, Pentium II processor, has symmetric L1 instruction cache and data cache, both of which have 16KB. However, asymmetric L1 caches are more suitable to improve the performance. Figure 14 depicts the different impacts of L1 instruction cache and data cache on IPC. Increasing the size of L1 instruction cache contributes 60% performance improvement when the size ranges from 8KB to 128KB.

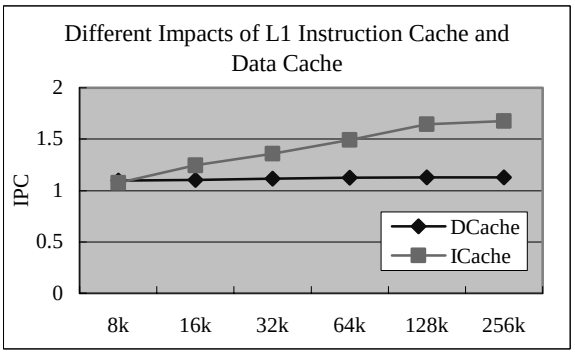


Figure 14. Different Impacts of L1 Caches on IPC

Based on the above observations, when we combine the results of Table 2 and Table 3 with those in Figure 2, we propose that these micro kernels be put in a part of the instruction cache which is not replaced to make room for other instructions.

5. Conclusions

This paper has presented a micro-benchmark, ServBench, for use in benchmarking the micro performance of Web servers. All the benchmark programs are taken from the implementation of the most commonly used Apache Web server by measuring and profiling the server with realistic workloads. We do not consider network and disk I/O functions for the reason that there has already been extensive research in decreasing and optimizing the network and I/O latency. All the dataset for the benchmark are obtained from the realistic workloads. Then we port the micro-benchmark and Apache Web server to SimpleScalar simulation environment. We obtain execution time, branch prediction and cache miss results for the micro-benchmark as a function of various architecture parameters.

The average code size of ServBench is an order of magnitude smaller than SPECint. Both have similar instruction set characteristics. ServBench has smaller basic-block sizes and nearly half of the branches are taken and half not taken. This fact makes better branch predication mechanism and lower miss rate very important to the performance.

By comparing ServBench and SPECint, we observe that L1 instruction cache plays a more important role than data cache in improving micro performance. We prove this observation by porting and running Apache in SimpleScalar simulation environment. We find that instructions executed per cycle are increased by 60% when the size of L1 instruction cache increases from 8KB to 128KB. Asymmetric L1 caches help to improve micro performance greatly. We are able to achieve higher IPC by using asymmetric L1 cache and enlarging the length of

instruction fetch queue and adding more ALUs. However, 4 ALUs and an instruction fetch queue of length 8 are enough to enhance the micro performance.

References

[1] M. Almeida, V. Almeida, D.J. Yates, Measuring the Behavior of A World-Wide-Web Server, 7th IFIP Conference on High Performance networking (HPN), White Plains, NY, Apr. 1997

[2] Apache, http://www.apache.org/

[3] M.F Arlitt, C.L. Williamson. Web Server Workload Chracterization: The Search for Invariants, Proceeding of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, pages 126-137, 1996

[4] G. Banga, P. Druschel, Measuring the Capacity of a Web Server, Proceedings of the USENIX Syposium on Internet Technologies and Systems, Monterey, Dec 1997

[5] G. Banga, P. Druschel, J. C. Mogul. Better Operating System Features for Faster Network Servers, Proceedings of the Workshop on Internet Server Performance, Madison, WI, June 1998

[6] G. Banga, J.C. Mogul, Scalable Kernel Performance for Internet Servers Under Realistics Loads, Proceedings of 1998 Usenix Annual Technical Conference, New Orleeans, LA, June 1998

[7] P. Barford, M. Crovella, Generating Representative Web Workloads for Network and Server Performance Evaluation, Proceeding of the ACM SIGMETRICS'98 Conference, Madison, WI, 1998

[8] D. Burger, T.M. Austin, The SimpleScalar Tool Set, Version 2.0, Technical Report, Computer Science Department, University of Wisconsin-Madison, June 1997

[9] S. Glassman, A Caching Relay for the World Wide Web. WWW'94 Conference Proceedings, 1994

[10] N. Gloy, C. Young, J. Chen, M. smith, An Analysis of Dynamic Branch Prediction Schemes on System Workloads, Proceedings of the International Symposium on Computer Architecture, May 1996

[11] S.L. Graham, P.B. Kessler, M.K. McKusick, gprof: A Call Graph Execution Profiler, Proceedings of the SIGPALN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982

[12] GNU Unix Toolset. Information and binaries available at http://www.gnu.org/

[13] V. Holmedahl, B. Smith, and T. Yang, Cooperative caching of dynamic content on a distributed web server, Proceedings of 7th IEEE International Symposium on High Performance Distributed Computing (HPDC-7), Chicago, IL USA July 28-31, 1998.

[14] Y. Hu, A. Nanda, Q. Yang, Measurement, Analysis and Performance Improvement of the Apache Web Server, the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99), Phoenix/Scottsdale, Arizona, February 1999

[15] C. Huitema, Network vs. Server Issues in End-to-end Performance, Keynote Speech, Performance and Architecture on Web Servers (PAWS), June 2000

[16] R. Iyer, Exploring the Cache Design Space for Web Servers, Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'00), San Francisco, CA , April 2000

[17] S. Manley, M. Seltzer, M. Courage. A Self-Scaling and Self-Configuring Benchmark for Web Servers. Proceeding of the ACM SIGMETRICS'98 Conference, Madison, WI, 1998

[18] D. Mosberger, T. Jin, httperf---A Tool for Measuring Web Server Performance, Workshop on Internet Server Performance (WISP98), Madison, Wisconsin, June 23, 1998

[19] E. Nahum, T. Barailai, D. Kandlur, Performance Issues in WWW Servers, Proceedings of the international conference on Measurement and modeling of computer systems, 1999

[20] NetPerf, http://www.netperf.org/

[21] V. Pai, P. Druschel, W. Zwaenepoel, Flash: An Efficient and Portable Web Server, Proceedings of the USENIX 1999 Annual Technical Conference, Monterey, CA, June 1999

[22] V. Pai, P. Druschel, W. Zwaenepoel, IO-Lite: A Unified I/O Buffering and Caching System, ACM Transactions on Computer Systems, Vol. 18, No. 1, pp.37-66, February 2000

[23] R. Radhakrishnan, L.K. John, A Performance Study of Modern Web Applications, Euro-Par 1999, Lecture Notes in Computer Science, Springer, pages. 239-247, 1999

[24] SPECWeb99 Benchmark, http://www.spec.org/osg/web99

[25] Standard Performance Evaluation Corporation, SPEC CPU95 Version 1.10, August 21, 1995

[26] G. Trent, M. Sake, WebStone: the First Generation in HTTP Server Benchmarking, White Paper, Silicon Graphics, Feb 1995

[27] WebBench, http://www.webbench.com/, Ziff Davis, Inc. March 2000

[28] D.J. Yates, V. Almeida, J.M. Almeida, On the Interaction Between an OS and Web Server, Boston University Computer Science Department, Boston Univ., MA, Tech Report CS 97-012, July 1997

Session 3

Architecture Evaluation and Modeling

Compressibility Characteristics of Address/Data Transfers in Commercial Workloads

Krishna Kant and Ravi Iyer
Enterprise Architecture Laboratory
Intel Corporation

Performance Workloads in a Hardware Multi Threading Environment

Bret Olszewski and Octavian F. Herescu
IBM

A Processor Queuing Simulation Model for Multiprocessor System Performance Analysis

Thin-Fong Tsuei and Wayne Yamamoto
Sun Microsystems

Compressibility Characteristics of Address/Data Transfers in Commercial Workloads

Krishna Kant and Ravi Iyer
Enterprise Architecture Laboratory
Intel Corporation
{krishna.kant|ravishankar.iyer}@intel.com

Abstract

In this paper, we evaluate the compressibility of address and data transfers in commercial servers. Our proposed compression scheme is geared towards improving the efficiency of the transfer medium (busses, links etc) and increasing the performance of the system. We evaluate the potential of the basic compression techniques for two commercial workloads – SPECweb99 [21] and TPC-C[22] – based on trace-driven simulations. Based on the obtained results, we show that simple compression schemes show significant promise for reducing address bus width and moderate benefits for data bus width reduction. We also show the sensitivity of these performance benefits to the number of bits compressed and the size of the encoding/decoding table used. Additionally, we propose enhancements to the compression schemes based on (1) recognizing and utilizing data-type specific knowledge and (2) improving the replacement policy of the encoding/decoding table. The performance benefits of bus compression schemes with these enhancements are also presented and analyzed.

1 Introduction and Motivation

With the increasing demand for high performance systems, commercial servers are now designed with large caches and larger memory resources. In order to reduce the amount of resources needed, researchers have proposed compression as a solution. This includes compressed storage techniques [19, 26], compressed main memory [1, 23] to reduce memory resources needed and cache compression techniques [14, 27] to reduce the amount of cache space needed.

Our focus in this paper is to design simple compression schemes that helps reduce the amount of information transferred between the processor caches and the memory subsystem. This compression is primarily geared towards improving the performance and efficiency of the transfer medium (busses, links etc). Current generation front-end and back-end servers are typically bus-based, with each system bus supporting several processors. As we look at the potential for bus-based systems in the future, we find three key design pressures: (1) to scale in frequency comparable to processor frequency improvements, (2) to support larger bus widths for transferring larger cache lines within the same amount of time and (3) to continue to provide scalability with multiple processors. In this paper, we evaluate the benefits of simple compression techniques in reducing the amount of address and data transferred over the bus, thereby allowing for narrower busses, less cross-talk and potentially higher frequencies. It is important to note here that the benefit of compressing the information transferred between processor and memory not only applies to busses but also to point-to-point links that might replace busses in future servers. In the case of point-to-point links, the benefit of compression materializes as a reduction in the transfer latency (since much less data is transferred over the link).

Our main contribution in this paper is as follows. We present the basic premise of the compression techniques used for reducing address and data transfer lengths. We discuss the locality properties in address and data streams while running commercial workloads on servers. We evaluate the potential of the basic compression techniques for two commercial workloads – SPECweb99 [21] and TPC-C [22]. The evaluation is based on analyzing various traces collected on real systems. Based on these results, we show that simple compression schemes show significant promise for reducing address bus width and moderate benefits for data bus reduction. We show the sensitivity of these performance benefits to the number of bits compressed and the size of the encoding/decoding table used. Additionally, we propose enhancements to the compression schemes based on (1) recognizing and utilizing data-type specific knowledge and (2) improving the replacement policy of the encoding/decoding table. The performance benefits of bus compression schemes with these enhancements are also presented and analyzed.

The rest of this paper is organized as follows. Section 2

provides an overview of related work on compression schemes for servers. Section 3 presents the basic premise behind the address and data compression schemes and proposes potential enhancements. Section 4 provides an overview of our evaluation methodology covering details of workloads and traces. Section 5 presents the salient results and provides a detailed analysis of the benefits. Finally, section 6 summarizes the paper and presents a direction for future work in this area.

2 Background on Compression Techniques

In this section, we provide a brief overview of the past work in the compression area, particularly as it relates to increasing the system performance.

2.1 Disk/Memory Compression Techniques

Several papers including [19, 26, 13] have investigated the use of compressed storage to reduce paging. These *swap-space compression* techniques use a LRU main-memory cache to hold evicted pages in compressed form and intercept page faults to check if the requested page is available in the cache before a disk access is initiated. Such an approach could significantly improve the performance for applications that require a large amount of memory but do not manage their paging behavior. For example, scientific applications working with huge matrices or other data structures could benefit from this technique. Most of the server applications such as DBMS or web-servers carefully manage the paging activity and are unlikely to see any significant benefits from swap-space compression. In fact, the loss of memory to compressed cache and the overhead of compression/decompression could well deteriorate performance.

A slight extension of swap space compression is *compressed disk cache* which introduces a compressed cache for all disk I/O (including paging and file I/O). In addition to the evicted pages, this cache also stores the files evicted from the O/S or application managed file cache. Due to a much large space needed for the compressed disk cache, dynamic cache size adjustment similar to the one in [26] is necessary to ensure that the compressed cache does not starve the normal disk cache. Workloads that require a significant amount of I/O per transaction could benefit significantly from the compressed cache. Alternately, a compressed cache could allow the use of a lower performance disk subsystem and thereby significantly lower the total system cost. High density front-end servers could benefit from this because of their physical space and power limitations which do not allow large I/O subsystems.

An even more extensive use of compression involves storage of all main memory data in compressed form. Unlike the last two schemes, compressed storage of all data is much more complex since it introduces a new address space (the compressed physical address space) along with the issues of efficient storage and address translations. Furthermore, if memory accesses from bus masters are to be kept transparent, it is necessary to introduce a decompression cache where cacheline level accesses can be satisfied. IBM's MXT technology [1], takes this approach along with the added flexibility that certain regions of the memory can be set as compressed while others are uncompressed. Reference [23] describes the details of IBM's Pinnacle chipset that supports this technology. It compresses memory in 1 KB blocks and stores compressed blocks using up to four 256 byte segments. These segments could be located anywhere in the physical memory and are accessed using 4 pointers in the header part of each block. Although this generality avoids storage fragmentation, simpler schemes may be preferable.

2.2 Compression Algorithms

With the considerable interest in compression in the main memory, several studies have examined compressibility of main memory data and specialized compression algorithms. Reference [12] studies compressibility of many popular Unix desktop applications using both the traditional algorithms (e.g., LZW, Arithmetic coding [20]) and the X-RL algorithm invented by the authors [13]. The latter algorithm encodes 4 bytes at a time using partial matching of bytes and dynamic coding based on a small dictionary. It is claimed to be especially suited for small block sizes and hardware implementation. The authors show that this can be easily implemented in hardware to provide 4 bytes/cycle input rate to the compressor/decompressor. The adaptive LZ77 (de)compressor in IBM-MXT also achieves a similar rate, but appears more expensive to implement [4]. Since reads are typically lot more prevalent than writes, it generally helps to use *asymmetric algorithms* where the decompression speed is significantly higher than the compression speed. This property is particularly important to the implementation suggested in this document. Both LZ77 [28] and X-match [13] have this property, but LZ78 (or its variant LZW [25]) do not.

High order bits in basic data types often show a low entropy. This observation has been exploited in [26] which introduces a new compression scheme that examines 32 bits at a time and looks for redundancies in the 22 MSBs only. For certain types of data (e.g., integers, floating point), this could make the compression more effective than a traditional byte based compression. This leads to the following general observation: *when dealing with*

structured data, it is useful to do compression in the units in which the data items are accessed. In particular, in a relational database environment, it makes sense to base compression on data fields or sets of fields (e.g., keys), etc. Reference [7] addresses this primarily front the point of reducing the cost of I/O subsystem.

Compression of machine code presents a rather unique case. Viewed based on the bit/byte patterns alone, code does not provide too much opportunity for compression. However, it is possible to exploit the knowledge of about the instruction format in order to reduce the number of bits needed to identify the instructions. Reference [5] shows that it is possible to achieve compression ratios of 3-5 for code in this manner. Several other issues have also been examined in code compression including special algorithms that allow decompression starting with any cacheline [16] and procedure based compression where individual procedures are compressed as a unit and decompressed at run-time as needed [11]. Reference [15] reviews several code compression techniques and examines the performance of hardware managed code compression available in IBM PowerPC 405.

2.3 Cache Compression Techniques

Although much of the compression-related work has concentrated on compressed storage on disk and in main-memory, there are a few notable attempts to use compression within the processing core as well. In most cases, processor registers and L1 cache store uncompressed data and compressed storage is confined to L2 cache. Reference [14] discusses a selective compression scheme wherein the memory and L2 cache addressing is in terms of blocks of the size of one cacheline (as usual) except that a block may contain either one uncompressed cacheline or two adjacent compressed cachelines. This is driven by compressibility considerations — if two adjacent cachelines compress to no more than one block the storage is in compressed form, else it is in uncompressed form. The cache mapping scheme remains unchanged so that the cache storage of uncompressed lines is resolved in the usual way. For a compressed line however, the sets corresponding to both lines (i.e., with least significant address bit being both 0 and 1) are examined. The scheme uses a small decompression buffer between L2 and L1 which acts just like an intermediate cache. Eviction of a modified line from L1 that is a part of a compressed block in L2 results in decompression of the whole block, modification of the relevant part, recompression and writeback to L2. The main memory storage scheme used in [14] is rather simplistic and remains in the units of pages of normal size and half-size. A half-page is used if all blocks within a page are compressed. Additional bits are used in page tables to handle two different page sizes and to identify the compressed/uncompressed nature of individual blocks within a page. The paper shows detailed simulation results for SPEC95 benchmark to confirm the reduced miss rate and read/write traffic in the core.

Compression at the L1 level has also been considered. Reference [27] considers a scheme similar to the one above, i.e., two adjacent cachelines are mapped to either of the two adjacent sets if it is to be stored in compressed form. The main difference is that the compression is not at the level of cachelines, but instead for individual "items" (e.g., 32-bit words) in a cacheline. This compression is based on the premise that a majority of accesses are to a small set of values (e.g., 0, 1, starting address of a large array, substring of spaces, etc.) and thus can be easily replaced by an index into a table containing "frequent values". For this reason, the technique is also known as *frequent value compression*. The major problem with this scheme is the determination of most frequent values and the latency of additional table lookup.

3 Our Focus: Compression of Address/Data Transfers

The nature of the information can often be exploited for reducing the number of bits needed in the representation. For example, reference [6] makes the observation that the addresses appearing on the address bus show considerable locality which can be exploited for reducing the number of address lines. This is done by only transmitting high order bits of the address and obtaining the low order bits from an encoding/decoding table. Reference [3] proposes a similar scheme for the data lines based on the locality in data values. In particular, based on technical workloads, the paper claims that 16 LSB data lines carry 90% of the information contained in 32 bit data items. These studies were done for uniprocessor systems with very small caches and also for technical workloads only. In this paper, we start by verifying the effectiveness of such a scheme for commercial workloads as well as for multiprocessor servers. Additionally, we propose various enchancements to the compression technique and evaluate their effectiveness over the base scheme.

3.1 Compressing Address Transfers

We start by describing the basic scheme for compressing address bits for processor-memory transfers. If successive memory accesses are mostly concentrated within a small region, the high order address bits will change infrequently and need not be transmitted every time. Instead, a dynamic encoding scheme could put the high-order bits in an encoding table (or cache) and transmit only the table index for later "hits" in the table. The first time around,

the high order bits are transmitted so that an identical decoding table can be built on the other side as well without any special information transfer. Figure 1 illustrates this scheme with a 64 entry encoding/decoding table incorporated into a conventional (base) system as an example. The base system with support for 36-bit addressability is shown in Figure 1(a). In Figure 1(b), the system with address bus compression is based on 18 low-order bits and 18 high-order bits. The low order bits are transmitted directly. The high order bits are looked up in a 64-entry encoding table. If the entry is found (table hit), only a 6-bit index to the decoding table need be transmitted for the high-order part. Thus a 24-bit address bus suffices. A miss will, of course, require data transfer over 2 cycles. (Compressed and uncompressed transfers can be distinguished by a special treatment of table entry 0.) Thus, if a high table hit ratio can be achieved, the compressed bus provides 33% reduction in width over the original bus with only a small performance penalty.

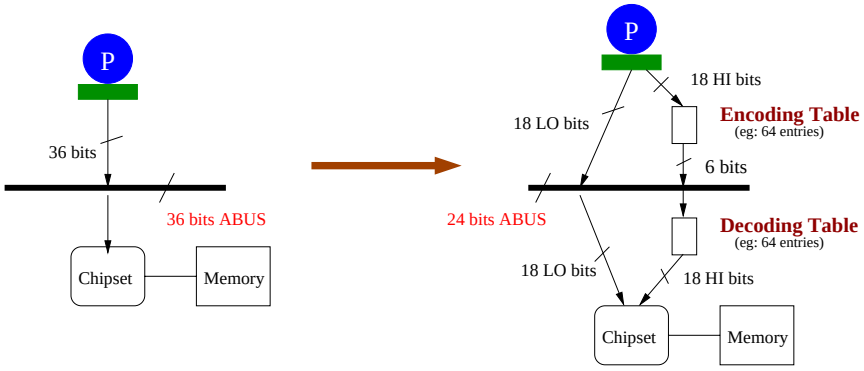
3.2 Compressing Data Transfers

A scheme similar to the address transfer compression scheme can be envisioned for compressing data transfers over the data bus as well. One major difference, however, in the data transfer case is that each cache block needs be divided into further smaller chunks before being encoded or compressed. However, choosing the right chunk size is not an easy task since the data accessed from the memory subsystem can be of various types (integer, pointer, floating point, code etc.) and lengths. The compressibility characteristics are very much dependent on the data type. For example, large values are rare for integers. This implies that the high-order bits for integers are usually all 0's or 1's, thus making integers a good candidate for compression. Pointer data typically shows considerable locality — long jumps or successive references to data items that are far apart are rare. However, this locality is in the virtual address space; it will be preserved only if the physical memory allocation is reasonably contiguous. Floating point data (single precision *float* or double precision *double* types) are usually poor candidates for compression except for generally very small exponents and frequent values like 0 or 1. Code data type also offers poor compressibility unless the knowledge of instruction set is exploited (which may be expensive for on-line use). Finally, the character string data offers moderate compressibility if the string length can be determined.

The main virtue of data type idenfication is that it allows certain optimizations that would be unavailable if all data were to be treated identically. For example, it makes sense to use the same number of low-order bits for both addresses and pointer data, and one may even consider a single encoding table for these. Similarly, if the double and/or code datatypes offer very poor compressibility, it might be preferable not to encode them at all since they would only pollute the encoding table. Even if all data is handled identically, it is interesting to find out what kind of hit ratios various data types get. In current architectures, data type information is not available for bus transfers (except for code fetch vs. data read/write); therefore, the identification must be based entirely on bit-patterns and could not be accurate. Fortunately, for our purposes, an accuracy of 80-90% is good enough. A good bit-pattern based identification depends on several factors including frequently used data types, machine addressability (32-bit or 64-bits), and compiler characteristics. Our method applies to the code generated by most C-compilers for most 32-bit machines. The method assumes implicitly that 8/16 bit integers and float datatype occur only sparingly. For simplicity, we consider data in 32-bit chunks only, which means that character and "short" integer data aligned on smaller boundaries will be missed. The details of our method are as follows.

- **Code Data Type** – Code fetches are easy to identify correctly since code fetches appearing on the bus usually possess a different request id than regular data reads.
- **Integer Data Type** – Here we check if the 8 MSBs are 00 or FF (in hex) in a 32-bit word. Note that if the 32-bit word contains 2 "short" (or 16 bit) integers, they will be regarded as single 32-bit integer. The scheme will work for 64-bit integer representations as well provided that the actual values of these variables rarely exceed 2^{31}.
- **Text Data Type** – This is identified by checking whether in each 32-bit word, *every* 8th MSB is 0 and *at least one* 7th MSB is 1. This is basically looking for 7-bit ASCII characters, at least one of which is a printable character.
- **Double/Float Data Type** – If two contiguous 32-bit words are neither Integer not Text, we look at first 5-bits to recognize small positive or negative exponents in the standard IEEE floating point format. If recognized as such, the data is declared as double. Note that this scheme will recognize two contiguous floats as a double — beyond this, no effort is made to recognize floats.
- **Pointer Data Type** – If a 32-bit word does not fall in any of the above categories, then the data type is assumed to be a pointer.

The above method needs some changes for 64-bit machines since the pointer data now occupies 64-bits. Most machines use (and are likely to use for quite some time)



(a) Base System

(b) System with ABUS compression

Figure 1: Illustration of Basic Compression Scheme (eg: for Address Transfers)

much fewer than 64-bits for addressing. For example, Intel's IA-64 architecture uses only 44-bits, which can address 16 TB of memory. This information along with the fact that the top 8 or more bits are unlikely to change much can be exploited for identifying pointer data more directly and compressing it more efficiently. If the compiler allocates 64-bits by default for integers, one could exploit this too, but since the top 32 bits will almost always be all 0's or all 1's, dealing with 32-bit chunks will work just as well.

4 Workload Traces, Tools and Methodology

Our evaluation methodology relies on bus traces collected on real systems running commercial workloads such as SPECweb99 and TPC-C. Here we present an overview of the two benchmarks and the trace configuration. We also describe the trace-driven tools developed to simulate the encoding/decoding table and its performance benefits. Finally, we present the scope of the studies undertaken.

4.1 Workloads and Traces

For this study, we used bus traces that were collected on a web server running SPECweb99 and on database servers running TPC-C.

SPECweb99 [21] is a benchmark that attempts to mimic a web server environment. The benchmark setup uses multiple client systems to generate aggregate load on the system under test (a web server). Each client (mimicing browsers) makes static GET, dynamic GET or a POST request to the server. The get requests always retrieve a file from the server, but for dynamic gets, the file may be appended with some dynamically generated data. The retrieved file comes from a file-set with access characteristics defined via a 2-level Zipf distribution. Some studies on cache/memory access characteristics of web workloads can be found in [17, 8, 9]. For our study, we used SPECweb99 traces from a 2P web server since a dual-processor configuration is more characteristic of systems used in this market segment.

TPC-C [22] is an online-transaction processing benchmark that mimics a parts ordering database system. The transactions include entering and delivering part orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Some studies on the cache/memory access characteristics of OLTP workloads can be found in [2, 18, 10]. For our study, we used TPC-C traces from a 4P system. In addition to the above mentioned 32-bit systems, we also used a TPC-C trace from a 4P 64-bit system for understanding the impact of compression schemes on 64-bit architectures.

4.2 Simulation Tools and Studies

To perform the compressibility analysis, we developed a tool called **S**imulator for **CO**mpressed **T**ransfers (SCOT). SCOT basically provides mechanisms to simulate encoding/decoding tables for both address & data transfers. The management of data in the encoding/decoding table is an important aspect that needs to be addressed. Our aim in this paper is only to show the potential of compression schemes for address/data transfers. We plan to investigate detailed design issues for the encoding/decoding table in the future after determining that there is some performance potential to be taken advantage of. So, in this early evaluation, we consider the encoding/decoding table to be a fully associative array. We determine the performance potential (in hit ratio) of this scheme as a function of the encoding size (number of high-order bits that are compressed) and the size of the encoding/decoding table. We then study the impact of different replacement strate-

gies (FIFO, LRU and our proposed MLRU policy). While the first two are well understood, our proposed MLRU scheme is based on giving lower priority to blocks that are first brought in. The plain LRU scheme can be thought of in terms of a stack with the blocks ordered in terms of their access time. So, in the LRU scheme, when a block is first placed on the stack, it is given the highest priority. The MLRU scheme alters the priority of incoming requests by using a parameter that controls where they are placed among the list it is placed. For the results presented here, an incoming block is placed only 25% above the lowest priority block. As a result, if the block is not accessed soon after it is entered into the stack, it is replaced. This tends to filter out one-touch references [24] and thereby leads to better performance than plain LRU.

5 Preliminary Results and Analysis

5.1 Compression Characteristics of Address Transfers

The performance benefits from the basic compression scheme for address transfer in 32-bit systems is shown in Figures 2 and 3. The x-axis in these figures denotes the table size (on a log scale) and the y-axis represents the hit ratio in the encoding/decoding table. For these results, a FIFO replacement policy is assumed. It is found that 85-90% hit ratio is easily achieved by using a table size of 64 entries and 20 high order bits (out of a total of 36 bits). The total number of address lines needed in this case is 22 (16 low order bits + 6 bit table index), which corresponds to a 39% reduction. This verifies that the compression scheme continues to work for commercial server workloads and for much larger caches than those considered in the original reference [6].

We experimented with a 64-bit system configuration as well. Here we chose the top 24 bits (out of a total of 44 bits) for compression. Only one trace (TPC-C with 16 GB of memory) was available in this case, so the results shown in Figure 4 should be considered as somewhat preliminary. We found that a 85-90% hit ratio needed a larger table size of 256 entries, which gives a savings of 16/44 = 36%. One reason for not getting better results is the larger database size and throughput for this configuration (consistent with 16 GB installed memory) which may reduce locality. The other possibility is the O/S (Windows 2000 for this configuration vs. NT4.0 for the 32-bit configuration). The O/S could affect the locality in two ways: (a) Size and locality of the O/S code itself, (b) Virtual to physical address mapping, since the mapping could perturb the inherent program locality significantly. In fact, an important point to keep in mind is that if address compression is to be exploited fully, the mapping algorithms should also be redesigned so that they don't significantly increase the entropy of the high order address bits. In this sense, our results are conservative and would improve with better address mapping schemes.

In order to attempt to increase the address table hit ratio further, we explored the use of better replacement policies. We tried three schemes: FIFO being the base, the well-known LRU scheme and our proposed MLRU scheme as described in Section 3. Figure 4 presents the benefits of the MLRU scheme for 64-bit systems. As shown, the MLRU scheme improves the address table hit ratio by roughly 5%. We will show the effectiveness of the MLRU scheme for data transfers also in the following subsection.

5.2 Compression Characteristics of Data Transfers

For data transfer compression in 32-bit machines, we used 16-high order bits of each 32-bit word for compression (thus avoiding any special treatment of double datatypes). Note that this choice makes the pointer data and address compression mutually compatible since 16 lower order bits are used in each case. Figure 5 illustrates the results for TPC-C. For these results, all data types were encoded through the same table. This coupled with the fact that 16-high order bits are used for all data types means that the data type identification is not required and does not affect the results. (The identification was still done in order to obtain data-type specific hit ratios). It is seen from Figure 5 that assuming a maximum reasonable table size of 256 entries, the achievable hit ratio on 32-bit systems is only 73% with the FIFO policy and 78% with MLRU policy, both of which are perhaps not very attractive.

Windows O/S allocates integers as 32-bit quantities even on a 64-bit machine. Therefore, for data transfer compression in 64-bit machines, we continued to compress 16 high-order bits of successive 32-bit words for all data-types except pointers. In order to make the treatment of pointers compatible with addresses, we use top 44 bits of 64-bits for compression. Of course, first 20 bits out of these are guaranteed to be zero since the addresses are only 44-bits long. (This obviously requires identification of pointer data based on bit patterns.) Figure 6 illustrates the results for TPC-C. It is seen that the performance in this case is significantly better than it was on 32-bit systems. In particular, for a 256-entry encoding/decoding table, we get a hit ratio of roughly 87% with FIFO policy and 90% with MLRU policy.

In order to understand the above results, we next look at the frequency of access to the different data types and the hit ratio for each data type as a result. The data type access frequency is shown in Table 1. From the 64-bit data shown, we find that the access to integers dominates the

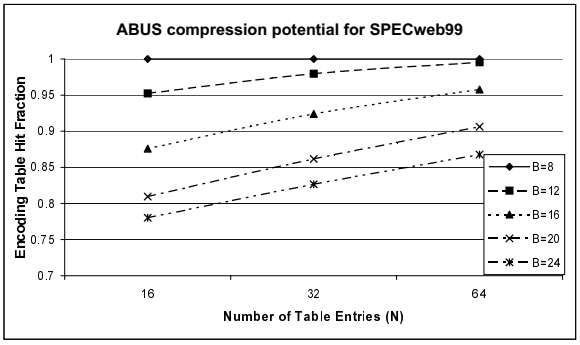


Figure 2: SPECweb99 Addr Compression: Varying Block Size, Table Size

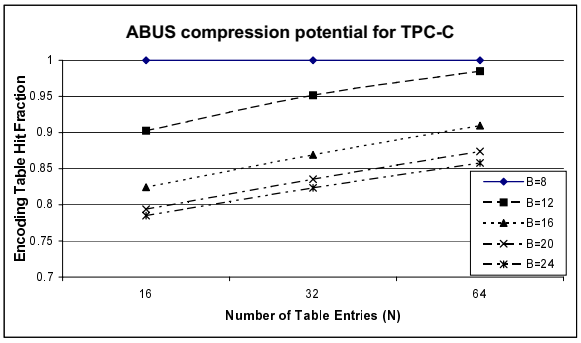


Figure 3: TPC-C (32-bit) Addr Compression: Varying Block Size, Varying Table Size

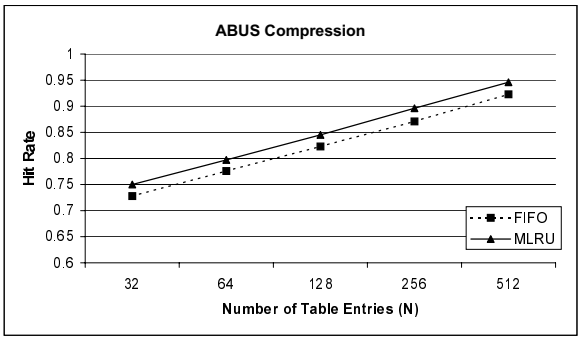


Figure 4: TPC-C (64-bit) Addr Compression: Varying Table Size, Replacement Policy

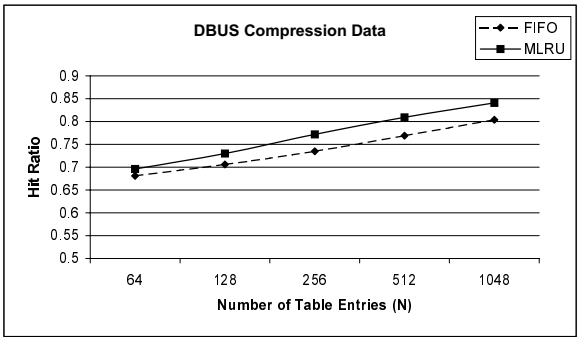


Figure 5: TPC-C (32-bit) Data Compression: Varying Table Size, Replacement Policy

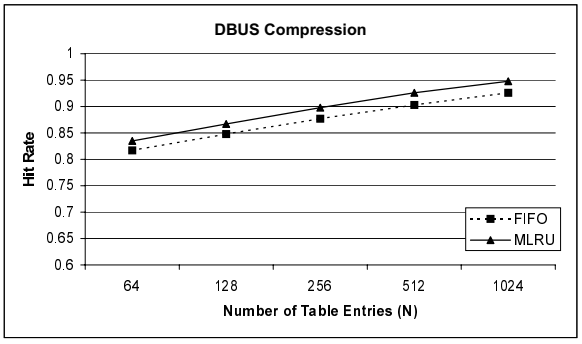


Figure 6: TPC-C (64-bit) Data Compression: Varying Table Size, Replacement Policy

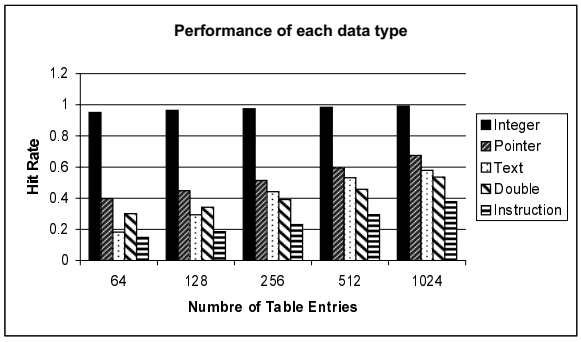


Figure 7: TPC-C (32-bit) Data Type Specific Hit Rates

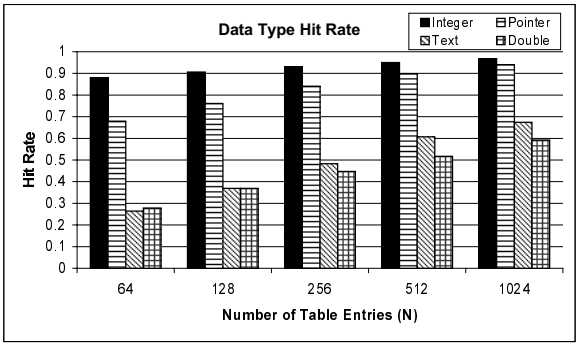


Figure 8: TPC-C (64-bit) Data Type Specific Hit Rates

Data Type	32-bit systems	64-bit systems
Integers	61.43%	91.35%
Pointers	19.80%	1.98%
Text	13.33%	6.50%
Double	0.34%	0.18%
Instruction	5.10%	NA

Table 1: Data Type Access Frequencies in TPC-C

hit ratio obtained. From the 32-bit data shown, we observe that the most frequently accessed data type in TPC-C is integers (with 62% access probability) and the least frequently accessed data type is double (with 0.34% access probability). Figures 7& 8 show the hit ratio for each data type. It is seen that integers have very substantial locality (and hence hit ratios), pointers have a reasonable locality, and code/double have the least amount of locality.

From Figure 7 it is clear that while pointer data type is the second most frequently accessed, its hit ratio is signficantly lower than the integer hit ratio. We hypothesize that the main reason for the low hit ratio in 32-bit systems is that the top 16 bits of pointers still have quite a bit of entropy. In the 64-bit case, Figure 8 shows that pointer data type also provides a good hit ratio; although in the considered case, the percentage of pointer references is rather small, which means that the overall performance is dominated by the integer hit ratios. As other 64-bit traces become available, we plan to do further analysis and see if the good data hit ratio obtained for this trace persists.

Although the address/data bus compression schemes appear suited only for data transmission over a bus/link, they are really no different from the compression schemes used for storage. The encoded data retains all the information needed for reconstructing the encoding table and decoding the data (except for the static parameters such as the table size, which can be remembered elsewhere). It is crucial, however, that the information be decoded in the same order as it was encoded. This is not an issue with bus transfer; in other contexts, this property can be enforced by dividing data into blocks so that each block is handled separately and involves a separate encoding table. Obviously, small block sizes will result in very little compression in general.

6 Summary and Future Work

In this paper, we evaluated the compressibility of address and data transfers in commercial servers. We started by presenting the basic premise behind simple compression schemes that use encoding/decoding tables. We showed that address transfers in web servers as well as OLTP servers show signficant potential for compressibility (from 85 to 90% hit ratio) for a reasonable table size (64 entries). We also showed that data transfers in 32-bit systems show only moderate potential (roughly 75%) even for a reasonable table size (256 entries). We also showed that we can increase this compressibility potential by improving the replacement policy (MLRU) and by using data type specific optimizations.

In the future, we would like to explore this technique by delving into the implementation issues (set-associative tables, bus protocol issues) and associated performance evaluation. We would also like to incorporate the hit ratio data into a system-level performance model in order to evaluate the overall performance impact for commercial workloads. We would also like to expand our current suite of commercial workloads by including integer/floating point workloads (SPEC2000), JAVA workloads (SPECjbb) and e-commerce workloads (TPC-W).

References

[1] B. Abali, H. Franke, S. Xiaowei, et.al., “Performance of hardware compressed main memory”, The Seventh International Symposium on High-Performance Computer Architecture, 2001, (HPCA2001), pp. 73 -81

[2] L. Barroso, et al. ”Memory System Characterization of Commercial Workloads”, *Proceedings of the 25th Annual*

International Symposium on Computer Architecture, pp3-14, June 1998.

[3] D. Citron and L. Rudolph, “Creating a wider bus using caching techniques”, Proc of first Intl symposium on high performance computer architecture, Jan 1995, pp 90-99.

[4] D.J. Craft, “A fast hardware data compression algorithm and some algorithmic extensions”, IBM Journal of R&D, Vol 42, No 6.

[5] J. Ernst, W. Evans, et. al., “Code compression”, Proc of 1997 SIGPLAN Conf. on Programming Language Design and Implementation, June 1997.

[6] M. Farrens and A. Park, “Dynamic base register caching: a technique for reducing address bus width”, Proc. of 18th annual Intl. symposium on computer architecture, May 1991, pp 128-137.

[7] B.R. Iyer and D. Wilhite, “Data compression support in databases”, Proc of 20th VLDB conference, Santiago, Chile, 1994, pp. 695-703.

[8] R. Iyer, ”Exploring the Cache Design Space for Web Servers,” Invited Paper, International Parallel and Distributed Processing Symposium (IPDPS’01), May 2001.

[9] R. Iyer, ”Performance Implications of Chipset Caches in Web Servers,” submitted to an international conference, Oct 2001.

[10] R. Iyer, et al., ”A Trace-driven Analysis of Sharing Behavior in TPC-C”, *2nd Workshop on Computer Architecture Evaluation using Commercial Workloads*, 1999.

[11] D. Kirovski, J. Kin, W.H. Mangione-Smith, “Procedure based program compression”, Proceedings of 30th annual IEEE/ACM International Symposium on Microarchitecture, 1997, pp. 204 -213

[12] M. Kjelso, M. Gooch, S. Jones, “Empirical study of memory-data: characteristics and compressibility”, IEE Proceedings on Computers and Digital Techniques, Vol 145, No 1, Jan. 1998, pp. 63 -67

[13] M. Kjelso, M. Gooch, S. Jones, “ Design and performance of a main memory hardware data compressor”, Proceedings of the 22nd EUROMICRO Conference, Beyond 2000: Hardware and Software Design Strategies, 1995, pp. 423 -430

[14] Jang-Soo Lee, Won-Kee Hong, Shin-Dug Kim, “An on-chip cache compression technique to reduce decompression overhead and design complexity”, Journal of systems Architecture, Vol 46, 2000, pp 1365-1382.

[15] C. Lefurgy, E. Piccininni, T. Mudge, “Evaluation of a high performance code compression method”, Proceedings. 32nd Annual International Symposium on Microarchitecture, 1999, MICRO-32, pp. 93 -102

[16] H. Lekatsas and W. Wolf, “Code compression for embedded systems”, Proc. 35th design automation conf., 1998.

[17] P. Mohapatra, H. Thanthry and K. Kant, ”Characterization of Bus Transactions for SPECweb96 Benchmark,” 2nd Workshop on Workload Characterization (WWC), Oct 1999.

[18] P. Ranganathan, K. Gharachorloo, et al., “Performance of Database Workloads on Shared Memory Systems with Out-of-Order Processors,” Proceedings of the Eighth International Conference on Architecture Support for Programming Languages and Operating Systems, Oct. 1998.

[19] S. Roy, R. Kumar, M. Prvulovic, “Improving system performance with compressed memory”, Proceedings 15th International Parallel and Distributed Processing Symposium, Apr 2001, pp. 630 -636

[20] K. Sayood, *Introduction to Data Compression*, 2nd edition, Morgan Kaufmann, 2000, chapter 5.

[21] ”SPECweb99 Design Document,” available online on the SPEC website at http://www.specbench.org/osg/web99/docs/whitepaper.html

[22] Transaction Processing Performance Council, TPC BENCHMARKTM C Standard Specification, http://www.tpc.org/, Jan. 2000.

[23] R.B. Tremaine, T.B. Smith, et. al., “Pinnacle: IBM MXT in a memory controller chip”, IEEE Micro, March-April 2001, pp 56-68.

[24] U. Vallamsetty, P. Mohapatra, R. Iyer and K. Kant, ”Improving the cache performance of network intensive workloads”, Proceedings of the International Conference on Parallel Processing, 2001.

[25] T.A. Welch, “A technique for high-performance data compression”, IEEE Computer, pp 8-19, June 1984.

[26] P.R. Wilson, S.F. Kaplan and Y. Smaragdakis, “The case for compressed cache in virtual memory systems”, Proc. USENIX 1999.

[27] Jun Yang, Youtao Zhang, R. Gupta, “Frequent value compression in data caches”, Proc. of 33rd annual IEEE/ACM Intl Symposium on Microarchitecture, 2000. MICRO-33, 2000, pp. 258 -265

[28] J. Ziv and A. Lempel, “A universal algorithm for data compression”, IEEE trans. on information theory, Vol IT-23, No 3, pp 337-343, May 1977.

Performance Workloads in a Hardware Multi Threaded Environment

Bret Olszewski **Octavian F. Herescu**

IBM Corp.
Austin, TX

Abstract

This paper describes the benefits of hardware multithreading (HMT) on IBM's eserver Pseries systems on commercial workloads. The HMT mechanism takes advantage of today's considerable gap between the speed of the processor and memory to increase throughput by overlapping memory fetches with computation. We analyze the performance improvement by measuring 5 commercial benchmarks: SDET, Netperf, OLTP, Websphere and SFS, which were run using the same hardware configuration with and without HMT enabled. We also discuss the characterization of this technique compared to single-threaded processors and determine that by enabling HMT, the throughput improvement is in the range of 10-20%, confirmed by the CPI measurements. Using the hardware performance monitor we study many other hardware counters that factored into the improvement, including HMT specific ones.

1 Introduction

System design trends have in recent years concentrated on increasing instruction level parallelism (ILP). Highly superscalar designs, when coupled with aggressive memory prefetching mechanisms, have proved highly effective in integer and floating point applications. However, ILP gains in large footprint commercial codes have been virtually stalled.

According to Moore's Law, the processor's speed doubles every 18 months. That has held true for the last 36 years. Unfortunately, the speed of the memory system has not increased at the same rate. The main improvements of the memory components have been in density and manufacturability and not in performance. This has resulted in increased amount of memory per system at reduced cost. But each new generation of technology makes the latency to memory become a greater factor and that has led to increasingly complex system structures to maximize overall hardware performance.

One method to improve system performance is to overlap memory accesses with other instructions. This can and has been done in a number of mechanisms. The object of this paper is a mechanism called hardware multithreading (HMT).

2 Hardware Multithreading

HMT is a technique for tolerating memory latency by utilizing cycles in the CPU that normally would be stalled waiting on memory accesses [1]. On a system, such as the server Pseries 660 6M1, that contains relatively large on chip L1 caches and a large L2 cache, the ratio of loads and stores to memory accesses is reduced to less than one hundred to one. This number is still high because it means

that the processor spends an average of about 1.0 cycles of waiting on memory for each instruction running commercial workloads. *Figure 1* shows an estimate of the time the CPU stays stalled.

Considering that the system used in this paper uses an RS64-IV processor, which can execute up to four instructions per cycle, the time spent waiting for memory is far more than the time the instruction spends in the processor.

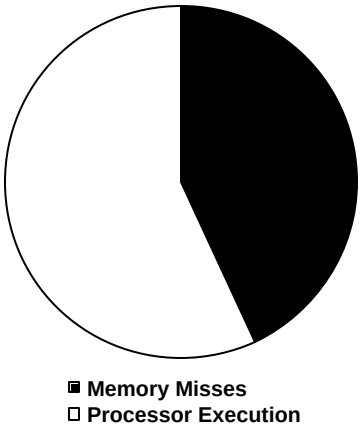


Figure 1 - CPI breakdown for uniprocessor

Hardware multithreading provides a mechanism for improving the overall system throughput by overlapping memory accesses with the execution of other instructions. This means that there will be multiple "active" threads per processor, which requires the replication of the processor-architected registers for each thread.

Cost savings, when compared to adding another physical processor, exist because replication is not required for the majority of the processor logic, such as: instruction cache, data cache, TLB, instruction fetch and dispatch mechanisms, branch units, fixed-point units, floating-point units and storage-control units.

The sharing of so many units can take its toll on the performance of some workloads, one of which will be described in this paper. The RS64-IV processor has two contexts per physical processor. At any processor clock, only one of the logical processors (contexts) can have instructions in the execution pipeline. The other context is inactive. See *Figure 2*. The physical processor switches between the contexts at a fairly rapid rate giving the software the impression that it is dealing with two distinct processors. The fact that the processor used as an example in this paper has a four-stage pipeline allows a thread switch penalty of only three cycles.

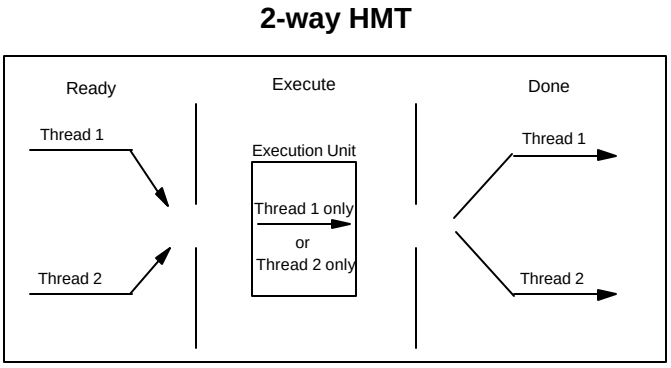


Figure 2 - HMT Only one thread active at any time

One of the requirements to make this method efficient and to achieve the improved performance is that the switch time between threads must be shorter than the latency of the event that triggered the switch. In our implementation the minimum penalty for an L1 miss is nine cycles. The thread switch penalty is only three cycles. *Figure 3* shows the switch timing for a L1 cache miss with an L2 cache hit. The thread switch penalty and the penalty for

TLB misses or instructions cache misses are comparable. As mentioned above, the processor switches between threads on selected events. The mechanisms to switch between contexts are controlled by the operating system and the events are selected by the thread switch control register (TSC). Three basic classes of switch events have been defined:

?? hardware events (cache misses and virtual memory translation misses).

?? software hints (software is allowed to have different priorities assigned to the logical processors).

?? time out (one logical processor cannot starve the other).

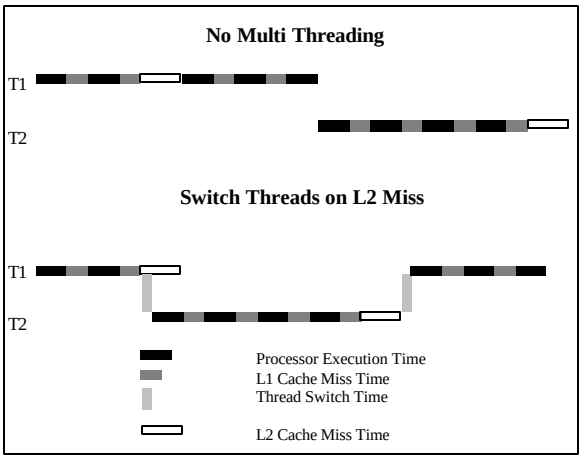


Figure 3 - Switching Penalty

The two logical processors may have different priorities. If the first thread on a processor is busy, the second will wait for an event that will trigger a logical processor switch. Hardware events are the main triggers of switching between the two logical processors. One of the most frequent causes for a switch is a cache miss. A cache miss occurs when the active thread executes a load, store, or instruction fetch and the referenced data is not found in the cache. The cache miss causes the processor to switch to the other thread. The thread that was waiting on the secondary logical processor can now execute. But usually the execution is interrupted when the data needed by the first processor becomes available. The most unfortunate case is when the second thread also has to wait for data from memory. A switch can also occur when the hardware priority of the logical processors changes. We mentioned that each logical processor has its own hardware priority. Normally the hardware priority is 2 (medium priority) but software may switch the priority to a higher value (3 for example) using the ***or r3,r3,r3 noop*** or to a lower priority using the *or* ***r1,r1,r1 noop*** command. The hardware will switch immediately between threads when the priority is changed. The high priority **noop** can only be used in kernel mode.

To avoid starvation, one more event that can trigger a switch was introduced: time out. The hardware maintains a clock counting the number of cycles that the current thread has been executing. When the time out is exceeded, the hardware will switch to avoid thread starvation.

The HMT mechanism is particularly appropriate for the RS64 microprocessor. The logic overhead required to implement HMT is less than 10% of the chip size. The cost in cycle time to implement HMT is also small, less than 10%. A much more deeply pipelined processor could not so efficiently switch between threads due to its more complex state. On deeply pipelined processors, it is expected that simultaneous multi-threading (SMT), a mechanism

where multiple threads execute instructions at the same time, will prove the preferred solution. See *Figure 4*. SMT combines HMT with superscalar processor technology and allows multiple threads to issue instructions each cycle [3].

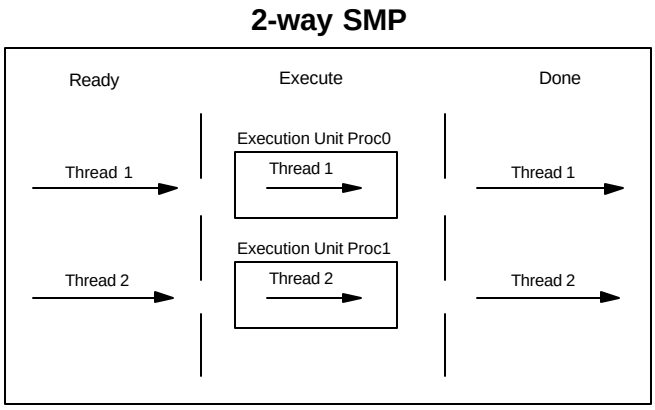


Figure 4 - SMT Two threads could be active at any time

3 AIX changes to implement HMT support

The first AIX version that supported HMT was 4.3.3. The HMT capabilities of AIX may be enabled or disabled by the *bosdebug* command followed by *the bosboot command*. The new mode will only be activated after a reboot. When HMT is fully enabled, the OS will see the system as having twice the number of processors; though the software which reports hardware configuration (e.g. lscfg) will report the number of physical processors. It is important to note that, unlike chips with independent cores [2], a hardware failure in the CPU core will disable both logical processors. Though the number of logical processors is doubled, each processor will typically have the performance of a little more than one half of the non-HMT processor.

There were very few changes required within AIX to support HMT. These changes were implemented in system initialization, dispatching, idle process, and locking. The system initialization changes include modifications to enable the second logical processor, which required changes to start the secondary threads. Also, all the areas in the kernel that have to be aware of the difference between the real number of physical processors and the number of logical processors, were modified. For example, the operating system allocates only one set of mbuffs per physical processor saving memory.

By executing the command *netstat -m* the user will only see mbuffs for processors with even numbers: 0, 2, 4, etc. The operating system was modified to use software hints for low priority operations like the idle loop and lock spinning. Since HMT gives software the illusion of two logical processors per physical processor, each logical processor has operating system process management resources. This includes an idle thread per logical processor. The idle thread is optimized to run at a low hardware priority. This optimizes performance at low CPU utilization: if one logical processor is idle, the full resources of the physical processor are available to the other logical processor. Additionally, the areas associated with locking were modified. Every time the active thread aquires a kernel lock, the hardware thread priority is increased to high. When the lock is released, the priority is switched back to medium. This optimizes the cross section of critical code. Another performance improvement comes from the optimization to adjust hardware priority to low for code spinning on locks. This effectively yields the physical processor to the alternate thread. These two

changes tend to improve overall throughput because the cpu is kept busy during otherwise idle cycles.

The current hardware implementation restricts that external I/O interrupts are presented on only one logical processor per physical processor. This characteristic could affect some specific workloads, which will be detailed in the next chapters. The usage of thread priority software hints allows AIX to ensure that sufficient resources are available to service interrupts thus minimizing the shortcomings of the above asymmetric characteristics.

4 Performance Workloads

While quite a number of benchmarks have shown improvements when running over HMT enabled systems, in this paper we will present only a few representative workloads and describe the kind of benefits they get from HMT.

I. SPEC SDM SDET - Software Development Environment Throughput, is a benchmark that simulates a multiuser environment in which the users are executing randomly ordered scripts; basically, sets of AIX commands in shell scripts. One of the main reasons we chose this workload is that this benchmark stresses three main components of an Operating System: file management, processor management and the virtual memory manager. Over the years since SDET's first release, it has become clear that the benchmark's primary use has not been in evaluating how many work units a system performs per time, but how well a system responds under stress, and whether a set of operating system modifications will result in performance improvements.

II. Netperf - is a benchmark that can be used to measure various aspects of networking performance. Currently, its focus is on bulk data transfer (streaming) and request/response performance using either TCP, UDP, or the Berkeley Sockets interface. While this benchmark is now part of the public domain, IBM has developed a derivative tool which is more tightly integrated with the capabilities of the AIX operating system.

III. Online transaction processing workload (OLTP) simulates a complete computing environment, where a population of users executes transactions against a commercially available 64-bit database. The benchmark is centered around the principal activities (transactions) of an order-entry environment. This workload has the largest memory-footprint as well as executable size of the commonly executed performance workloads.

IV. Websphere eBusiness Benchmark - is an IBM internal benchmark used to evaluate performance using the IBM Websphere Application Server software product. The benchmark emulates the operation of an online brokerage firm. One or more network attached HTTP clients drive a server running servlets which use Enterprise Java Beans (EJBs) that service requests and access a relational database. The benchmark thus exercises the operating system using HTTP services, Java, and a relational database.

V. SPEC SFS97_R1 V3.0 - is a standard benchmark for evaluating the performance of a Network File System (NFS) file server. The variant used in this benchmark measures the NFS version 3 protocol over UDP. One or more clients drive a mix of

operations to a file server. The benchmark requires a large disk and network configuration to reach peak performance.

All of the above benchmarks were run on an RS/6000 model pSeries 660 6M1 with 8 processors running at 750 MHz, IBM SSA disks and varying amounts of RAM. The workloads are quite different in their consumption of CPU time resources. *Table 1* shows that the workloads are all CPU limited in the configurations used for this paper. The large amount of CPU time consumed in the operating system for netperf and sdet, which are used as operating system tests, is a marked difference from the more real-world OLTP and Websphere benchmarks. The NFS protocol is optimized to execute within the AIX kernel context, thus all execution time is system.

Workload	%User	%System	%I/O Wait	%Idle
Sdet	33	65	1	2
Netperf	1	99	0	0
OLTP	85	15	0	0
Websphere	66	33	0	0
SFS	0	98	2	0

Table 1 - non-HMT CPU Time

5 Hardware Measures

To facilitate an understanding of the benefits and effects of HMT on system performance, the aforementioned workloads were evaluated with HMT both enabled and disabled. The most important metric defining the benefit of HMT is the increase in workload throughput obtained while using it. *Chart 1* shows that the improvement observed is typically on the order of 10% to 20% for these workloads. The improvement in throughput is directly related to the increase in instructions executed per unit time by the processors. This metric is typically defined as the cycles per instruction, or CPI.

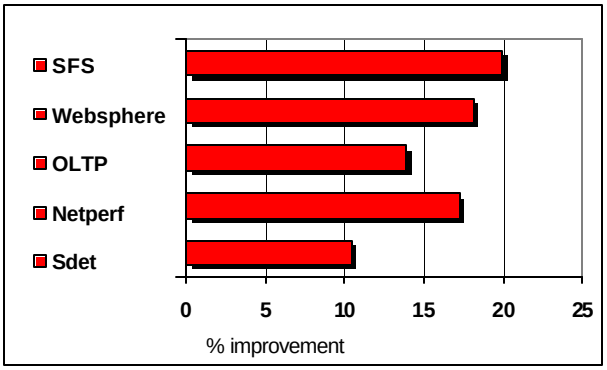


Chart 1 - Throughput Improvement with HMT

The hardware performance monitor can be used to assess the efficiency of the microprocessors with and without HMT enabled. There are a single set of hardware counters when HMT is enabled. The counters can be directed to count events for one or both hardware threads of execution for each physical processor, which allows us to directly count the total number of instructions executed by both hardware threads over a known number of cycles. The implementation of two threads sharing cache and translation resources would be expected to increase the frequency of cache misses. Since the L1 instruction and data caches as well as the TLB (translation look-a-side buffers) are relatively smaller than the L2 caches, they should see a larger increase in miss rates. As expected, the CPI with HMT enabled is lower, allowing higher throughput. This relative improvement in CPI is shown in *Chart 2*. Note that the improvements in CPI do not always scale linearly with throughput. This is due to

software efficiency as well as slight changes in the amount of idle time in the workload.

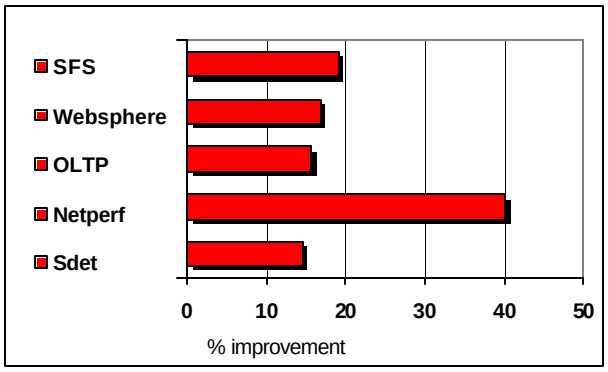


Chart 2 - CPI Improvement with HMT

Chart 3 shows the increase in instruction TLB (ITLB) miss rates with HMT. It would be expected that the largest increases would be in workloads with a multiple executables or very large executables.

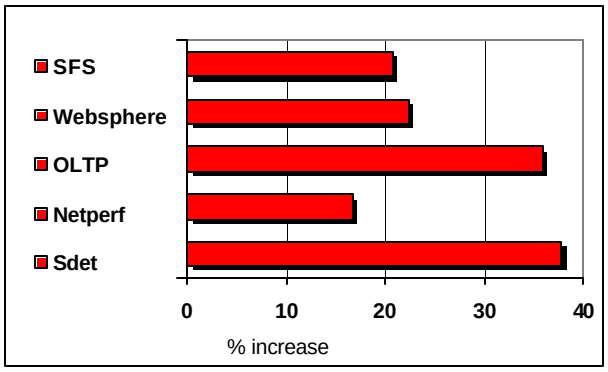


Chart 3 - Increase in ITLB per inst with HMT

This proves true for Sdet and OLTP, but the small increase in Websphere is surprising given its mix of large sized executables. As expected, netperf sees the smallest increase in ITLB miss rates, as the workload is dominated by kernel activity and a single small executable.

Chart 4 shows the increase in data translation lookaside (DTLB) misses per instruction with HMT. In this metric, the OLTP workload with its enormous data footprint and high multiprogramming level sees the greatest increase in DTLB misses.

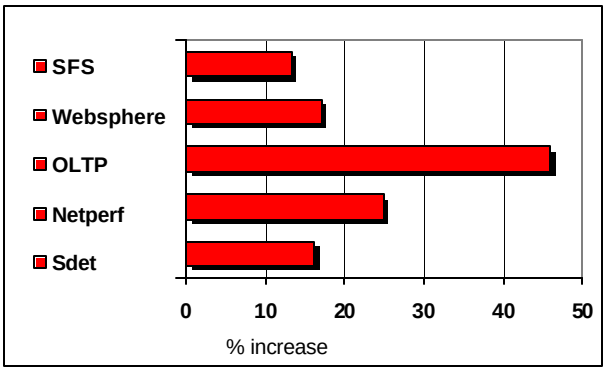


Chart 4 - Increase in DTLB per inst with HMT

The netperf workload, with a number of processes with small but essentially identical process address spaces, is penalized for aliasing of those addresses within the DTLB.

Chart 5 shows the increase in level-one instruction cache misses (IL1) with HMT. The increase in the IL1 cache miss rate for Netperf is somewhat misleading, as the absolute miss rates in each case are fairly low. This particular workload has a small instruction cache footprint. The increase in IL1 miss rate for Sdet is expected, as a large number of different executables are context switching within the shared IL1 cache with HMT. The increase observed in SFS has not been explained. Since the executable code is shared by all of the software threads, it is odd that instruction cache miss rate increases so greatly.

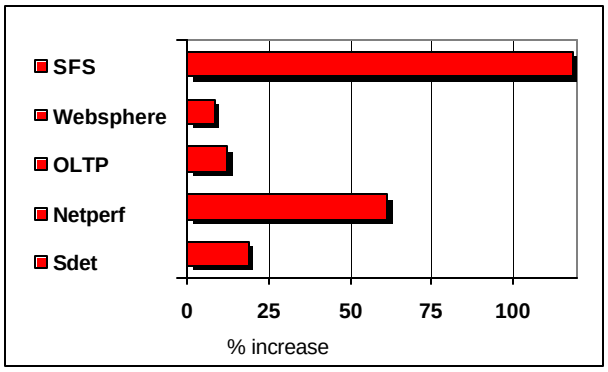


Chart 5 - Increase in IL1 per inst with HMT

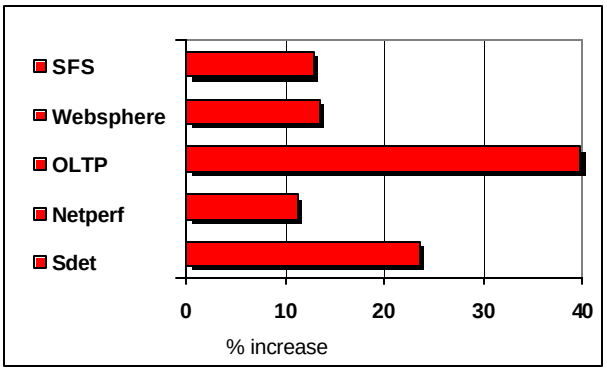


Chart 6 - Increase in DL1 per inst with HMT

Chart 6 shows the increase in level-one data cache misses (DL1) with HMT. Once again, the OLTP workload suffers the greatest increases with HMT for reasons sited with it's DTLB miss rate. While the change in L1 miss rates are not typically crucial in performance for commercial workloads, a exception has been consistently observed running the SPEC CPU2000 SPECint_rate2000 workload. This workload runs multiple copies of the exact same programs in lockstep, which tends to result in extreme thrashing of the L1 data cache. On this workload, we consistently see a decrease in performance using HMT. While cache thrashing is fairly unlikely in complex environments, it may appear on systems with low multi-programming levels and long running jobs. The increase in miss rates for L1 caches and translation is indeed important, but the effect on the L2 is usually much greater due to memory latencies. *Chart 7* shows in the studied benchmarks the increase is much lower on L2 miss rates due to the large size of the L2 caches on the system used. The decrease in L2 miss rate for Websphere is due to an actual decrease in the L2 misses resolved from another L2 cache. This is actually a benefit of HMT sharing L2 cache between two logical processors. The reason we see a decrease in the L2 cache misses ratio for netperf is that while the actual number of misses increased, the ratio miss/instructions decreased because the machine has more idle cycles.

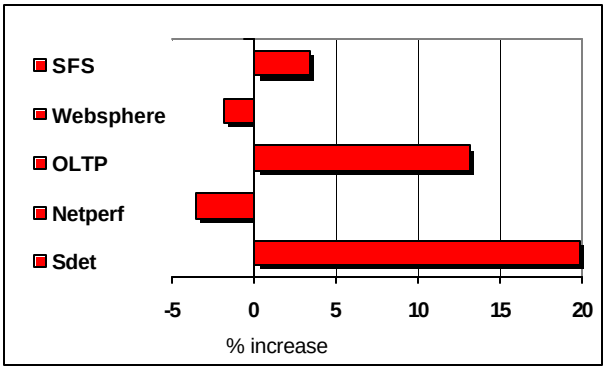


Chart 7 - Increase in L2 miss rates with HMT

We used the same workload for netperf as in the non-HMT case, but with HMT the system will be less utilized showing significant scaled throughput due to the increase in the number of CPUs . This somehow abnormal behavior is detailed in section 7. Though obvious, we must mention that adding HMT cannot improve performance if memory bandwidth

becomes constrained. Experiments on systems with bandwidth limitations have show very small gains in throughput. Another less studied issue is the effect on L2 miss rates with smaller L2 caches. There was concern that large footprint caches, such as OLTP, with very large instruction cache footprints, would scale poorly due to L2 cache thrashing. We were able to do some measurements on a one-processor 660 6F1 system using the OLTP workload. This system configuration has only a 2MB L2. The results of this measurement showed that the increase in L2 miss rates with a 2MB L2 were consistent with those observed on systems with 8MB L2 caches. A determination of miss rate increases below this size is difficult, as the 2MB L2 is still more than large enough to fully contain the instruction footprint. Since the instruction footprint is the most cacheable part of the workload, it seems likely the L2 miss rate increase will be larger with smaller caches.

6 HMT Specific Metrics

The hardware performance monitor also permits analysis of the HMT mechanism. These metrics provide a glimpse of how often there is switching between hardware threads as well as the causes of these switches. The rate of switching is a factor of cache and translation miss rates, software hints, and time-outs. *Chart 8* shows the average number of instructions between switches for the workloads. The hardware instrumentation also provides for counting the HMT thread switches, by the cause of the switch. *Charts 9,10,11,12, and 13* show the distribution of causes of most thread switch events for the three workloads.

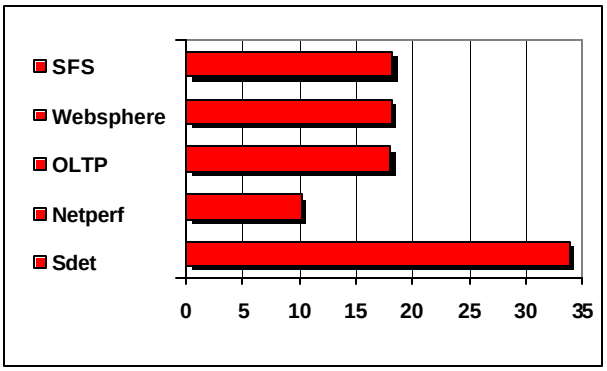


Chart 8 - Average instructions between switches

A few other thread switch cases, such as switch for L2 instruction miss, occur but are extremely infrequent.

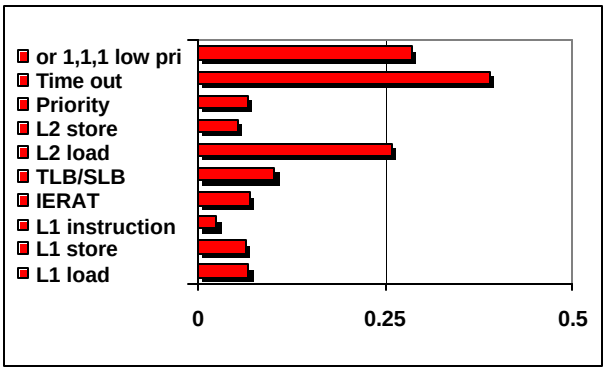


Chart 9 - Sdet HMT Fraction Switches

AIX configures the hardware to switch on all possible events. There are some circumstances when an L1 cache miss does not result in an immediate switch, which means the L2 switch events are not a subset of the L1 switch events.

A large fraction of switches are caused by the thread switch time-out. This time-out is set to a conservative value of 64 cycles. This value was selected to improve responsiveness for interrupt processing for high speed I/O adapters.

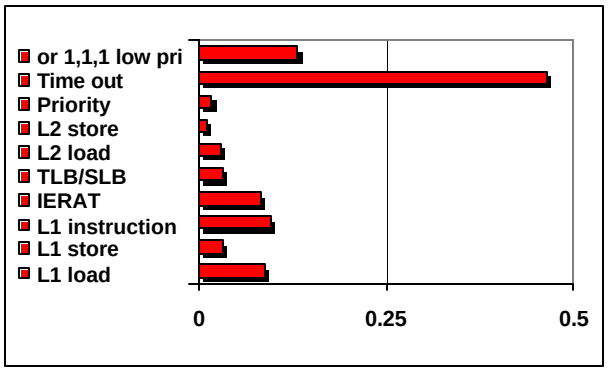


Chart 10 - Netperf HMT Fraction Switches

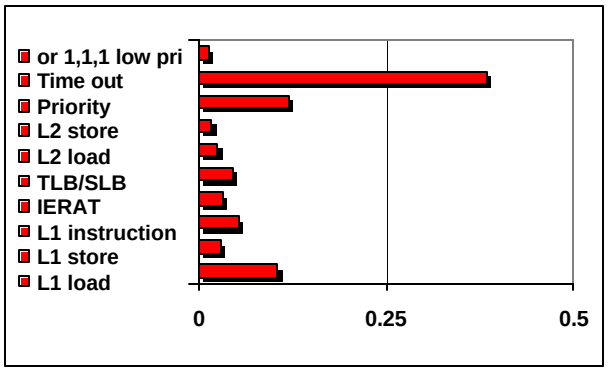


Chart 11 - OLTP HMT Fraction Switches

A higher value of time-out would improve throughput, though at the cost of responsiveness. The ***or 1,1,1 noop*** instruction switches are typically the result of the idle process or lock spin switches for low priority.

7 Performance Tradeoffs

The nature of HMT performance enhancements assumes highly multi-programming workloads (meanning when a task is waiting another one is running). AIX' s implementation of HMT attempts to provide fairness between the threads that are executing. While this balance is very effective for transactional environments, it is not optimal for single threads that need to execute and complete in the shortest amount of time.

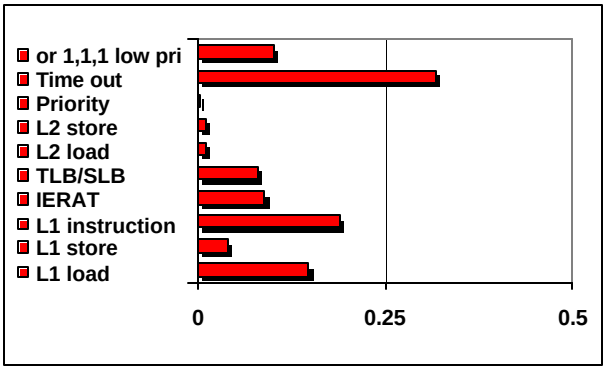


Chart 12 - Websphere HMT Fraction Switches

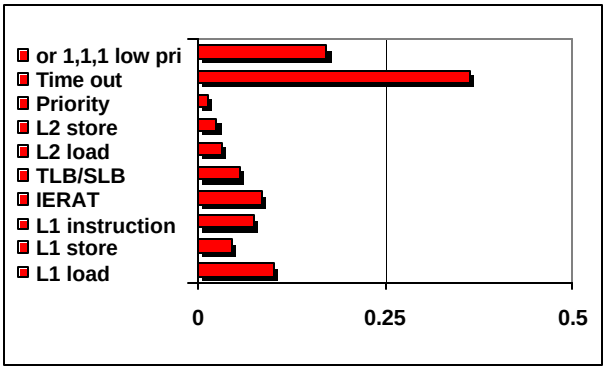


Chart 13 - SFS HMT Fraction Switches

No circumstances will allow a thread to have a shorter execution time in an HMT enabled environment. Let' s assume that there is only one thread that executes on a system that supports HMT. Whenever there is a cache miss, the system will switch to the other logical processor until the data for the miss becomes available. That means that there are two switches and each introduces a 3-4 processor cycles penalty. Had the system not switched, the running thread would not have been penalized. We saw the penalty in the time, using a business intelligence workload which measured the

run time per query. While the throughput increased when the queries were run in parallel, the time to run individual queries in isolation increased in a range from 2% to 20%. Since all the benchmarks we discuss in this paper ran for a predefined amount of time, this shortcoming of HMT could not be quantified. It is important to understand the type of workload that a system is most likely to handle and based on that, make a decision whether it is beneficial to enable HMT on the system. Thus, the benefits of HMT for highly transactional, highly multi-programming works must be weighted against the drawbacks for time-critical tasks.

Another side effect of HMT is a consequence of sharing the physical resources of one processor between the two logical processors. And that is an increase in variability in elapsed and CPU time to complete a task. AIX treats the two logical processors as equals and that means that the amount of resources available to one processor can, and is, affected by what is running on the other one. When HMT is fully enabled AIX behaves as if the system has doubled its number of processors. This increase could create scalability problems. Resource sharing also poses a challenge to the CPU monitorization tools. AIX determines the CPU utilization using a common sampling technique querying the tasks running on each processor 100 times per second. This implementation has each processor taking clock interrupts on average each 10 milliseconds. The hardware supports two mechanisms, one allowing the two logical processors to take clock interrupts in absolute time, the other to take them proportional to their execution time. Clearly, having 100 clock interrupts per second delivered proportionally to the two logical processors would improve the accuracy of CPU utilization, since the logical processors may proceed at different rates. But many tools assume, based on constants in header files, absolute clock tick counts. Thus AIX maintains its original behavior of each processor taking 100 clock interrupts per second. Note here that this problem will need to be resolved if, as in the mainframe world, multiple operating instances coexist on a physical processor (e.g. shared processor partitions). The problem with constant rate sampling is the capacity implied by the CPU utilization. In essence, if the system is 50% busy and performs a certain number of tasks per second, one may assume that approximately twice as many tasks per second can run on the system before the CPU is totally consumed. The above assumption does not take into consideration the other factors in scaling a workload: memory, I/O bandwidth, or workload scalability. When HMT is enabled on the system, with the sampling technique, the Operating System will sample both logical processors every 1/100th of a second. If only one thread is running on the system at a certain time, and it blocks for a cache miss, the system will switch to the other processor, which will be idle since there is only one thread executing in the system. The operating system will record the elapsed time on this processor as idle time. If another thread is running on the system, does the throughput increase linearly with the idle time that was previously observed? The answer is: not necessarily, since both threads could end up blocked on cache misses from time to time.

The way Netperf reports the system utilization is by averaging the CPU utilization of all the CPU's that the system detects. With HMT enabled, there will be twice as many processors available. When using Netperf , we used the same workload on both cases HMT and nonHMT. In the nonHMT case the workload used would make the CPU utilization be 100%. Using the same workload with HMT the CPU monitoring tool would report the CPU's as only 93% busy in average. There is no guarantee that even if there were no other constraints, like memory or I/O bandwidth, the system would be able to do 7% more work, which implies the CPU utilization observed with HMT is somewhat less precise as a measure of available capacity. So, if a certain workload requires a highly accurate CPU utilization, HMT is not recommended. In the case of Netperf, measurements showed that even if the workload was increased the throughput did not increase. In fact the throughput decreased while the CPU reached 100%.

8 Conclusions

Our analysis has shown performance gains of ten to twenty percent, on different types of workloads with high multi-programming levels are possible. It is important to understand that the gains are very much dependent on the type of the user's workload. Issues with the performance of long running critical tasks, workload scalability, cache thrashing and CPU utilization should be taken in consideration and quantified in order to determine the applicability of HMT in a specific user environment.

For future work we think that improvements could be done in both hardware and software by keeping these two goals in mind:

1) We need to have hardware that has a low switching cost, since that is the major overhead, and

2) To be able to provide good single-thread performance, therefore allowing applications with low parallelism to execute efficiently.

References

[1] J. Borkenhagen, R. Eickemeyer, and R. Kalla :A Multithreaded PowerPC Processor for Commercial Servers, IBM Journal of Research and Development, November 2000, Vol. 44, No. 6, pp. 885-98.

[2] Gulati, M., Bagherzadeh, N.: Performance Study of a Multithreaded Superscalar Microprocessor. 2nd Int. Symp. On High Performance Computer Architectures, February 1996, 291-301

[3] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen, *IEEE Micro*, September / October 1997, pages 12-18

Acknowledgements

Jim Van Fleet, Mysore Srinivas and Greg Mewhinney for their earlier work on HMT on AIX.

Steve Kunkel for RS64 hardware help.

Augie Mena, Sujatha Kashyap, Qunying Gao and Anthony Garcia for helping us collect the data.

Trademarks

A Processor Queuing Simulation Model for Multiprocessor System Performance Analysis

Thin-Fong Tsuei and Wayne Yamamoto
Sun Microsystems, Inc.
{thin-fong.tsuei,wayne.yamamoto}@sun.com

Abstract

This paper describes a processor queueing simulation model for a complex processor that aggressively issues instructions with the use of out-of-order, multiple issue and multithreading. The model is developed for memory and system evaluation of memory-intensive commercial OLTP (On-line Transaction Processing) workloads on large multiprocessor systems. Our approach differs from detailed cycle accurate and direct-execution simulations in that we do not simulate instruction execution. Instead, we take a high level view, as in analytical models, and model a minimal set of processor components to accurately generate the memory access traffic for system simulation. The model is validated with a cycle accurate simulator and is within 5% accuracy. Results on the effect of store burstiness and memory latencies on overlapping of cache misses and buffer sizing are presented.

1. Introduction

The performance requirements of many commercial applications can only be met with large, complex multiprocessor systems. Modeling and performance analysis for large systems are becoming increasingly important. With processors becoming more complex through the use of out-of-order multiple issue, multithreading, and chip multiprocessing, multiprocessor system models based on simple processor models no longer accurately represent the memory access workload to other system components. In particular, simple serial issue, block-on-cache-miss processor models cannot capture the characteristics of overlapping cache misses and processor stalls due to resource contention.

Cycle accurate processor simulators which simulate the micro-architecture of processors are essential and commonly used for research and design of processors [5]. However, this type of model, while accurate, is too complex and slow to be used effectively in large multiprocessor system models. Techniques based on direct-execution were developed to speedup execution and reduce complexity in processor simulations [2,3,4,8]. Since direct-execution simulation runs application code directly on a host machine, it may not be feasible for commercial applications that can easily require configurations of gigabytes of memory and disk space.

In analytical multiprocessor models, processors are generally modeled as a blackbox with emphasis on defining parameters that effectively characterizing the memory access arrival process. Parameter values are derived from measurements on existing system or obtained from other simulations. Very often, synthetic workloads are used to stress the system components to identify bottlenecks.

In this paper, we describe a processor queueing model for multiprocessor system performance analysis. The purpose of this processor model is for memory hierarchy and system design evaluation of memory-intensive commercial OLTP (On-line Transaction Processing) workloads on large multiprocessor systems. Our approach differs from detailed cycle accurate and direct-execution simulations in that we do not simulate instruction execution. Instead, as in analytical models, we view the function of a processor model is to provide the interactions between the processor and the rest of the system. We model a minimal set of pro-

cessor characteristics that accurately generate the memory access workload of a complex processor.

We note that a processor stalls due to contention for processor buffer resources in the cache hierarchy and other application characteristics in the processor pipeline. These buffer stalls are heavily dependent on the memory operations of the application and the system latencies. Consequently, we decouple the processor memory subsystem buffers from other pipeline operations. Our model treats the core pipeline as a black box and focuses on modeling the interactions among core pipeline and buffers in the processors memory hierarchy.

The processor queueing model is extensively validated with cycle-accurate processor simulation. While previous studies mainly validated using scientific workloads, we focus on commercial OLTP applications, and use the TPC-C (Transaction Processing Council) benchmark [13] with commercial database software as a workload. In the process of validating the model, we found that the model was sufficiently accurate for analyzing trade-offs in sizes and allocation policies for the buffers in the processor's memory subsystem.

We use the processor queuing model to investigate how stall cycles and the overlapping of cache misses vary with memory latencies. Pai et. al. [6] have found that read clustering, or burstiness, is required to achieve high overlapping on memory latency for SPLASH workload. Our results show that the stores of the OLTP workload are bursty. In addition, though bursty loads and stores improved the overlapping of memory access time, they are also more likely to fill up the buffers and cause the processor to stall. Proper sizing of the buffers and the cache architecture are important to reduce such processor stall time

The rest of the paper is organized as follows. Section 2 discusses some of the related work. Section 3 describes the model. Sections 4 and 5 present the validation and overlap study results. Section 6 discusses integration of the processor queueing model with the system model. Section 7 concludes the paper.

2. Related Work

There are several approaches to multiprocessor performance system evaluation.The most conservative approach is to do cycle-by-cycle simulation for the entire system from the processor to the interconnection network and the memory system [5,7]. This approach is detailed and has good accuracy, but is extremely slow. It is unpractical for evaluating a large multiprocessor system.

Direct execution is an alternate system simulation method with improved simulation time over the cycle accurate simulation [2,8]. As with the cycle accurate simulation, direct execution also models the actual execution of the instructions of an application on a processor. Direct execution, however, decouples the processing of the functional and timing components of instruction execution. The functional simulation is accelerated by executing the binary of an application directly on a host machine. The timing information is analyzed independently. Pai and Adve have done extensive studies with Ranganahan [6] and Durbhakula [2] on issues and methods to improve the accuracy of direct execution. Direct execution has improved the simulation time by a few times, but it still simulates many micro-architecture details. The size of the system that it can be applied to is limited.

Analytical modeling using mean value analysis (MVA) has been used in shared memory multiprocessor systems with processors that block on cache misses [15]. This technique is shown to be efficient and reasonably accurate for large systems. The new challenges in analytical modeling for today's superscalar, out-of-order processors are to model the overlap in memory accesses on cache misses and to estimate processor stall time. New parameters, such as fraction of overlap in load misses, number of outstanding misses, and probability of blocking due to full buffers, are defined to model the new memory access characteristics [1,10,14]. Except for [10], synthetic workloads are used for parameter values. In [10], the application parameters are obtained from a simplified processor simulation derived from DirectRSIM [2], including the blocking probabilities for a single buffer. Though the simulation is simplified and has achieved two orders of magnitude speedup over RSIM [5], it maintains the complexity needed to simulate instruction execution.

All the analytical studies assume that overlap and blocking parameter inputs are static and are unaffected by the contention delay at the rest of the system. In practice, blocking due to buffer capacity is a function of the latency and the contention delay at the system, and iterations between the processor simulator and the MVA system model are needed in high contention conditions. Our model can be used to derive the blocking parameter values quickly, and is much more

lightweight to iterate since we do not need to execute the entire application.

3. The Processor Model

3.1 Processor overview

The processor modeled in this study was designed to exploit both instruction level and thread level parallelism while running at a high clock frequency. It executes the SPARC™ instruction set and features deep pipelining with superscalar, out-of-order issue, speculative execution, and 2-way multithreading. Capable of fetching multiple instructions per cycle, the instruction fetch unit involves a two level adaptive branch outcome predictor and a branch target predictor. Fetched instructions are decoded, renamed and inserted into an instruction issue buffer (IIB). Instruction fetching is stalled once the IIB entries are filled. Once in the IIB, an instruction can be issued as soon as all its operands are available. Multiple instructions can be dispatched to functional units every cycle.

The cache hierarchy consists of three levels of cache: separate L1 instruction and data caches, a unified L2 cache, and a unified L3 cache. Loads and instructions that miss in the L1 caches are moved into a L2 load buffer (LB) which can process instruction/data read misses concurrently. When the LB fills up, processing of L1 cache misses is blocked and will eventually stall the processor. Read misses that miss in the L2 and L3 cache are sent out of the processor over a split transaction interconnect to the memory controllers.

A store remains in the IIB until it is the oldest instruction, then it is moved into the store buffer (SB). Once in the SB, stores are committed to the L2 cache in program order. Store misses requiring memory access can cause the SB to fill up which will back up into the IIB and may eventually cause the processor to stall.

The processor is highly speculative and uses both control and data speculation to boost performance. We use aggressive control speculation employing the branch predictor to fetch and execute instructions beyond multiple unresolved branches. On a misspeculated branch, instructions younger than the branch are flushed from the IIB and LB and instruction fetching resumes along the correct path. Data speculation is also employed to speculate on the value of loads. Correctly speculating on a load allows earlier dispatch of dependent instructions, misspeculation results in reissuing of these instructions.

3.2 Model overview

As discussed earlier, the goal of the processor model is to accurately reproduce the memory access traffic, that is, the memory access types and the rates at which they are generated.

As the instruction fetch has well defined and less complex blocking behavior, we decouple the processing of buffer management from the instruction fetching and core pipeline (Figure 1). In the processor modeled, as long as there is no mispredicted branch or instruction cache miss and the IIB is not full, instructions are fetched and installed into the IIB. Since we are mainly concerned with memory hierarchy and system design studies from L1 cache outward, we simplify the model by aggregating the fetch time with other pipeline operations and L1 cache access time into one service demand which feeds the IIB. This service time is the average execution time in an ideal L2 cache environment, including pipeline interlocks due to data dependencies, L1 cache misses, and branch mispredictions.

The contention for caches and buffer resources in the model adds delays and reduces the instruction execution rate. We model the instruction, load, and store misses as they travel between the buffers (IIB, LB, and SB), capturing the contention and

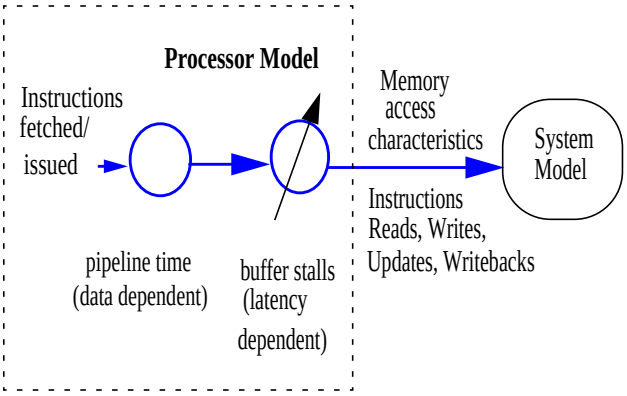


Figure 1. Breakdown of execution time in a processor

Table 1:Model Parameters

Application parameters	**Measurement of parameter value**
Service demand at the pipeline	Derived from cycle-accurate processor simulation with infinite L2 cache
Histogram of instructions per fetch	Derived from cycle-accurate processor simulation with infinite L2 cache
Loads per instruction Stores per instruction	Statistics collected from trace analysis
Histogram of consecutive store	Statistics collected from trace analysis
Cache misses: Level 1, 2 and 3 instructions, data reads, data writes, upgrades and dirty writebacks	Statistics collected from cache simulations and cycle-accurate processor simulation
System parameters	
Cache and memory latencies	Design specifications, values vary
Buffer sizes	Design specifications, values vary

eventual stalls. The contention delay for buffers is determined largely by cache miss rates and the memory latencies, the service time is relatively independent of the memory latencies.

Table 1 defines the model parameters. The service time for the core pipeline is the time to complete the instructions in a fetch when the L2 cache is infinite. The number of instructions in each fetch is determined by the application basic block characteristics and the processor fetch unit architecture. We use a histogram for the number of instructions per fetch. Other application characteristics include loads and stores per instruction, instruction fetch and data load and store cache misses per instruction, and the upgrades and writebacks per instruction. The histogram of consecutive stores can be used to model the store traffic more accurately

Figure 2 details the queueing model for the processor. The service demand for the core pipeline is modeled with two identical delay servers, one for each thread. At the end of the service time, *n* instructions are generated according to the instructions per fetch histogram. Every instruction resides in the IIB until it retires. Load and store instructions are assigned based on their per instruction probabilities. Instruction fetch and data cache misses are determined randomly according to their respective miss rate. On an instruction miss, the fetch process stalls until the missed instruction returns. A data load miss proceeds to the load buffer (LB) and proceeds to L3 cache and the system memory model when it misses in L2 and L3 caches respectively. For validation of the processor, we treat the system model as a delay server, i.e., a constant memory latency.

The in-order retirement of instructions is enforced by the IIB. If an instruction completes before older instructions, it remains in the IIB until it becomes the oldest. Stores remain in the IIB until they are the oldest and then they are moved to the SB where they must commit in order. Load misses are assumed to be independent and complete (i.e., can be removed from the LB) as soon as the data is returned. In general, load misses can be dependent. However, results of internal studies using an infinite buffer size [11,12] indicate that the effect of load-load dependencies on execution rate is low for OLTP workloads because of the random nature of the data accesses. For workloads where this effect is significant, the model can be modified to add a probabilistic measure of load-load dependencies.

While the highly speculative nature of the processor helps to boost performance, misspeculation results in more memory traffic due to cache pollution and possibly more stalls due to LB buffer contention. In our model, we account for misspeculation through a higher cache miss rate obtained from the detailed

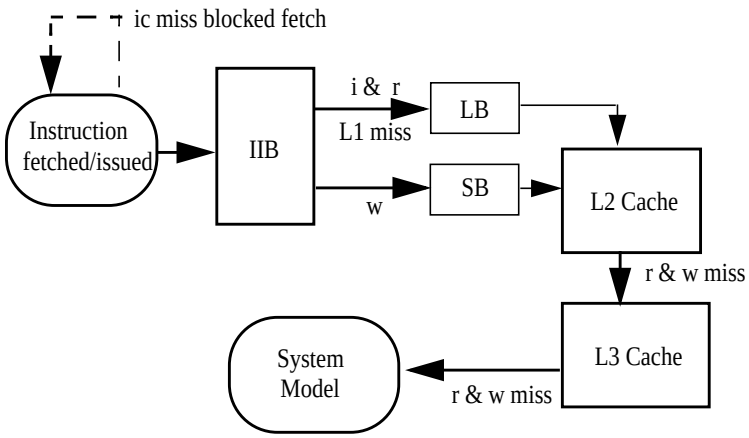


Figure 2. Processor Model

cycle accurate simulations. We do not model the LB and IIB flushing of the wrong path instructions on a branch misprediction since we compared the miss rates with and without branch misprediction and discovered that the differences were small. However, the model can be refined to flush the buffers with a branch misprediction probability.

The implementation of the model is relatively straightforward and is implemented using a commercial simulation package. The buffers are implemented as pools of numbered tokens. Tokens are retrieved from a buffer pool in sequential order and are put back whenever a buffer space is released. A store can be removed from the IIB once it is the oldest and acquires a space in the store buffer. When LB or SB is full, IIB can become congested and instruction fetches will eventually stall when the IIB is full.

We use two service centers to implement the pipeline cache. The first service center models the first stage of the pipeline. The second service center is simply a delay in the rest of the cache pipeline depth. We model alternate load and store queue scheduling, and first-come-first serve scheduling at the caches.

3.3 Obtaining Model Parameter Values

Table 1 also summarizes the measurement and estimation of the model's parameter values. The pipeline service demand is derived from the cycles per instruction when the L2 cache is infinitely large. This parameter and other application parameter values, such as the histogram of instructions per fetch and the loads and stores per instruction are derived from a single run of the detailed simulation model. An additional run is needed to derive the pipeline service demand for dual-threaded execution. The cache miss rates are obtained from cycle accurate simulations but could also be obtained using simple cache simulations. Both single and dual threaded cache simulations are run. Cache miss rates can also be varied independently for sensitivity studies. The histogram of consecutive stores was obtained empirically and was used to control the burstiness of stores. System parameters, such as buffer sizes, are obtained from processor design specifications.

The model application parameters are defined so that we can leverage on data from detailed trace-driven processor and cache simulation models that are always created in the development of new architectures. The histogram of instructions per fetch, and the values of load and store per instruction can also be estimated from other tools when a detailed simulator is not available. One such suite is the data prefetch analysis tools used in compiler technology [12]. These tools model the fetch unit and the branch prediction unit for data dependency analysis. The service demand can be estimated by static analysis, from older cycle accurate simulators or from existing machines.

Table 2:Processor Parameters

Instruction Issue Buffer	128 entries
L2 Load Buffer	32 entries
Store Buffer	32 entries
L2 cache	1MB, 4-way set associative
L3 cache	32 MB, direct mapped

Table 3: Comparison of IPC and number of outstanding Loads and Stores

	Single Thread Queueing/ Cycle Accurate	**Dual Thread** Queueing/ Cycle Accurate
IPC		
Trace-A	-0.43%	0.05%
Trace-B	9.03%	9.51%
Number of Outstanding at the System (Trace-B)		
Loads	2%	0.61%
Stores	3.2%	4%

4. Model Validation

4.1 Validation with Cycle Accurate Simulation

The model is validated with a cycle accurate processor simulation. The cycle accurate simulator models the processor and system architecture in fine detail. The simulator is trace driven and simulates execution on a cycle-by-cycle basis. The processor and system parameters were chosen to reflect typical systems in the near future. Table 2 summarizes the baseline processor parameters used in our validation. We use a constant 400 cycle memory latency for the baseline configuration.

For validation, we examined the IPC (Instructions Per Cycle), histograms of the buffers usage, and histograms of the outstanding memory loads and stores. We compare the results for two OLTP traces, Trace-A and Trace-B, collected from commercial database software. Trace-A is from a small configuration and Trace-B is from a larger configuration.

Table 3 shows the percent differences in IPC compared to the cycle accurate simulation for single and dual thread. The estimated IPC agreed very well for Trace-A for both single and dual thread. The queueing model estimates is about 10% more optimistic for Trace-B. The main differences between the two traces are the cache miss rates. The cache miss rates of Trace-B are significantly higher (> 50% more). To investigate the causes of the estimated differences, we studied how the two models compared in estimating the overlapping of the load and store misses. The findings for the dual thread results for Trace-B, which have the highest discrepancy between the two models, are discussed. The percent difference of IPC improves to 4.4% for dual thread Trace-B after we added store burstiness to the queueing model.

Table 4: Distribution of Loads and Stores at System for Dual Strand (trace-B)

Number at System	**Loads (percentages)**		**Stores (percentages)**	
	Cycle Accurate	**Queueing**	**Cycle Accurate**	**Queueing**
1	61.1	48.3	19.5	10.7
2	23.3	38.2	22.1	19.9
3	10.7	11.4	19.1	23.5
4	3.5	1.9	14.8	20.3
5 and more	1.4	0.2	13.5	26.8

Comparison of the number of outstanding loads and stores at the system (Table 3) for Trace-B showed that the queuing model is more optimistic. More loads and stores are able to access the memory concurrently and, hence, benefit more from overlapping. Table 4 shows the distribution of the number of cycles at which there are 1 to 5 and more number of outstanding loads and stores at the system. For loads, the queueing model has greater discrepancies at 1 and 2 loads, and is slightly optimistic in load overlapping. For stores, higher clustering when there are larger numbers of outstanding stores indicates that the queueing model is not blocking store misses often enough.

We further investigate the buffer usage. We find that the differences in the utilization of the buffers, as shown in Table 5, are within 6%. The utilization of SB is much higher than LB because we model a L1 write through cache. Nonetheless, the stores have a higher miss rate than loads at the L2 cache. Figures 3, 4 and 5 compare the buffer usage distribution for IIB, LB, and SB respectively. The IIB usage in Figure 3 shows that the two models agree well in the lower range but diverge at the higher end. The lower probability that the IIB is full means that the queuing model is less likely to be blocked.

Figure 4 shows that the queueing model has a higher LB usage at the lower range. In the queueing model, we omitted the effect of aliasing on multiple occupancy in the load buffers; as a result, the LB is less likely to have high concurrent occupancy than the cycle accurate model indicates. However, because the processor queueing model has higher instruction execution rate, more load misses arrive randomly at the LB. This explains the differences at a lower occupancy of LB and a higher average number of LB used. Since the usage of the load buffer is very low and a more refined model would have negligible improvement on the accuracy of the model, thus, we chose to keep the simple load buffer model.

The store buffer usage in Figure 5 shows that the cycle accurate simulator has a burstier arrival of stores. There is a higher peak when the buffer is full. The queueing model, on the other hand, is more uniformly distributed at higher occupancy. This is expected since the loads and stores are uniformly distributed in the instruction stream for the queueing model. We will discuss the effect of burstiness of stores in the sections 4.2 and 4.3 below.

Table 5:Buffer Utilizations for Dual Strand (trace-B)

Buffer Type	Dual Strand (trace-B)	
	Cycle Accurate	Queueing
IIB	39.6%	35.9%
LB	12.2%	17.2%
SB	44.3%	38.9%

4.2 Sensitivity Studies

We did sensitivity studies by varying the cache sizes and the store and instruction buffer sizes. The LB size is unchanged because its usage is low and has little effect. The L2 and L3 cache sizes are doubled, and the store and instruction buffer sizes are increased by two and four times. Table 6 compares the percent gains in IPC (Instructions per Cycle) predicted by the cycle accurate simulator with those from the queueing model. It shows that at the base cache and store buffer sizes, the difference in percent gain from the two

Figure 3. IIB Usage Distribution
(trace-B, dual strand)

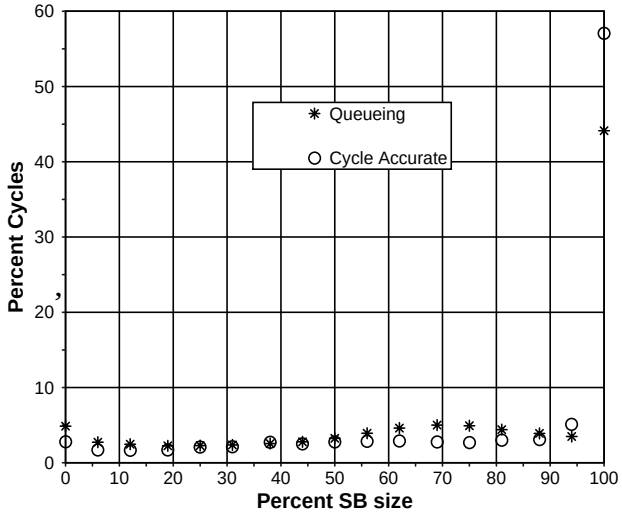


Figure 4. LB Usage Distribution
(trace-B, dual strand)

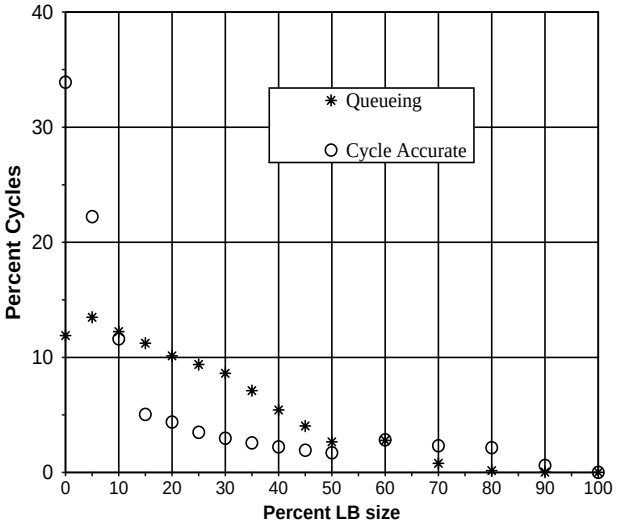
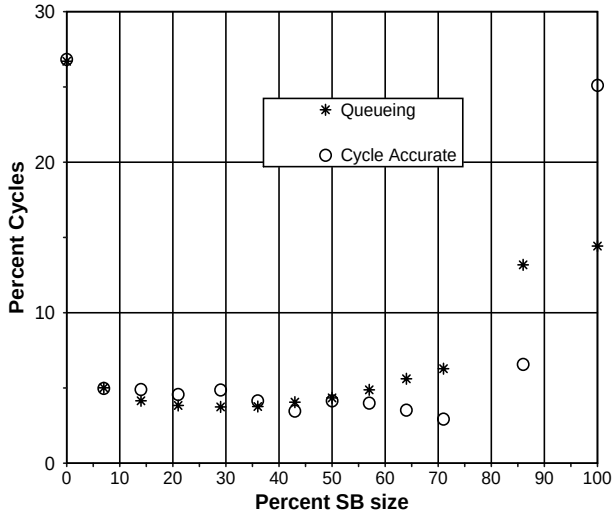


Figure 5. SB Usage Distribution
(trace-B, dual strand)



models is the largest. The cycle accurate model gains little by doubling the IIB size, while the queueing model gains a 16%.

As the contention on the store buffers is alleviated by either increasing the cache sizes and, hence, lowering the store cache miss rates or by doubling the store buffer size, the performance gain from architecture changes estimated from the two models are similar. The same design decision would have been made using either model.

From these results, we believe that the queuing model is not capturing the store buffer pressure as in the cycle accurate model. This difference becomes significant as more instructions are fetched when IIB is doubled. It is evident that the burstiness of store arrivals is important in identifying the correct size for the store buffer, which has significant impact on the performance.

Table 6: Sensitivity Studies Results

L2 (MB)	L3 (MB)	SB	IIB	%IPC gain Cycle Accurate	%IPC gain Queueing
1	16	32	64	base	base
1	16	32	128	5.9%	16.2%
1	16	64	128	11.8%	18.3%
1	32	32	64	2.9%	5.5%
2	16	32	64	14.7%	11.6%
2	16	32	128	23.5%	26.4%
2	16	64	128	26.5%	28.2%

Table 7: Distribution of Consecutive Stores

Number of consecutive stores	Random Assignment (%)	Burstiness Histogram (%)
1	68.1	41.3
2	24.9	28.7
3	6.0	0.0
4	0.9	0.0
5	0.1	0.0
6	0.0	0.0
7	0.0	16.5
8	0.0	13.5

4.3 Burstiness of Stores

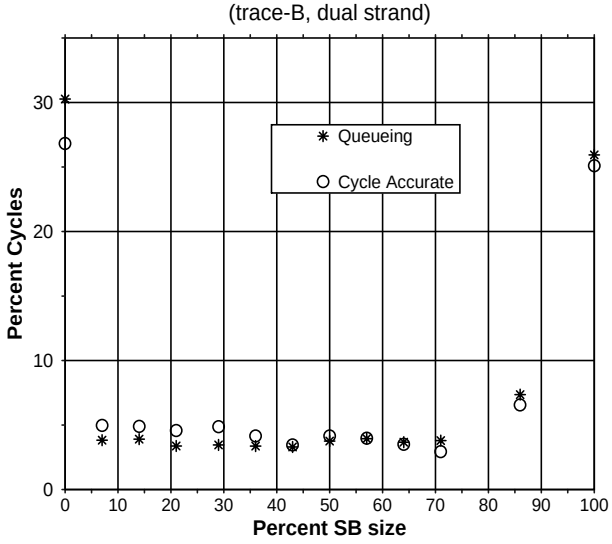
To verify that the burstiness in store arrivals is the reason that store buffer in the queueing model is not blocked as frequently as in the cycle accurate model, we modify the random assignment of stores in the queueing model to use a histogram to generate consecutive stores in each fetch. The average number of stores per instruction and the store miss per instruction remained unchanged.

Table 7 shows the distribution of consecutive stores in each fetch with random assignment and using an input histogram. Figure 6 shows that the store buffer usage of the two models agreed well. The difference in IPC and the average number of busy store buffers between the two models improve by 5% for the base configuration as shown in Table 8.

5. Overlap (Hidden) Miss Penalty Study

In an ideal buffer for an out-of-order, superscalar processor, one expects the longer the latencies, the more concurrent misses and greater the overlapping (or hiding) of miss memory access times. However, in practice, there are limited buffers. As latencies increase, buffers are held longer and the processor eventually stalls waiting for the buffers. After validating the queuing model with the cycle accurate model for different buffer sizes and latencies, we vary the memory latency to study the performance benefit of overlapping of misses as the latency increases, or alternatively, how to size the buffers with the memory latency.

Figure 6. SB Usage Distribution with Bursty Stores
(trace-B, dual strand)



To measure the benefit of overlap, we compare the hidden miss penalty (%Overlap) to that in a block-on-miss processor. To calculate %Overlap, we define the Load Miss Penalty as the number of cycles to resolve all load misses and Cycles Blocked as the number of cycles the processor was blocked due to memory operations. The %Overlap is the fraction of the Load Miss Penalty that are overlapped, i.e., the percent difference between the Load Miss Penalty and Cycles Blocked. In a blocking processor (i.e.,the processor stalls when there is a load miss) this overlap factor would be equal to zero.

Table 8: Comparison of IPC and SB Usage with Store Briskness

	Random Assignment	Burstiness Histogram
IPC Cycle Accurate/ Queueing	+9.5%	+4.4%
SB Usage	38.9%	42.7%%

Table 9: Processor Stall and Cache Miss Overlap for a Range of Latencies

Latency processor cycles	**%Cycles Blocked**	**%Overlap (i+r)**	**%Cycles SB is Full**
200	**34.9**	**61.7**	**11.8**
330	**45.8**	**47.8**	**28.8**
460	**55.7**	**31.5**	**38.6**
730	**69.0**	**4.8**	**48.2**
1000	**76.2**	**-11.3**	**52.5**

Table 9 shows the %Overlap factor as the memory latency in processor cycles is varied from 200 to 1000 cycles using the baseline buffer sizes and cache configuration. The results show that with a 200 cycle latency, as much as 61.7% of load miss latency is hidden. However, as the latencies increase, the %Overlap actually decreases. This is due to the increase in the %Cycles Blocked. From the results, we can see that the %Cycles Blocked doubles for a five fold increase in latency. Further examination revealed that the increase in %Cycles Blocked as the latency becomes longer is due to the SB being full. Table 9 shows the %Cycles the SB is full increases as the latency increases. In fact, as the latency increases, the blocking due to the SB full eventually results in the %Overlap decreasing until it finally disappears when the store buffer is about 50% of time full.

We did the same set of runs with one sixth store miss rate to reduce the pressure on the store buffer. The results shown in Table 10 show a definite decrease in %Cycles the SB is full as the latency increases. For a 200 cycle latency, stores have little effect and the percent of load misses that overlap is about the same as our previous result. For latencies greater than 200 cycles, the %Overlap is much better than our previous case. This shows that for the larger latencies, stores can have a significant effect on the %Overlap factor.

Table 10: Processor Stall and Cache Miss Overlap at a lower store miss rate

Latency processor cycles	**%Cycles Blocked**	**%Overlap (i+r)**	**%Cycles SB is Full**
200	**33.5**	**62.7**	**3.3**
330	**41.5**	**54.1**	**8.4**
460	**48.8**	**45.3**	**12.6**
730	**60.5**	**30.3**	**18.5**
1000	**67.6**	**21.0**	**22.1**

6. Integration with System Model

We are currently investigating how to integrate the processor queueing model into multiprocessor system performance evaluation. One approach is to link the processor queueing model with a system model for an end-to-end simulation, as is done conventionally. Another approach is to iterate between the processor queueing model and a system model. This is particularly useful when the two models are developed using different tools, or of different types (e.g., an MVA system model).

In the analytical model described in [3], a blocking probability vector is used to estimate processor stall time while waiting for a buffer. Our processor queuing model can generate similar blocking probabilities faster than an execution based model, and can quickly re-calculate probabilities as the latencies and contention at the system cause the blocking probabilities to change.

Many system performance issues can be analyzed with statistical synthetic workloads; however, there are some studies, such as buffer replacement algorithms and flow control at memory, that will require memory footprint with timing information. We are looking into augmenting cache simulation traces with timing information derived from the processor queueing model.

For the processor queueing model to provide accurate memory activities for a system model, the inputs to the queueing model must capture the application characteristics at the specific configuration of processors. In particular, cache miss rates and cache-to-cache transfer rates must reflect the larger database

and the greater number of processes executing in a large system.

A processor model is only a submodel in overall system performance analysis. The accuracy of the system submodel is as important to obtain an accurate overall performance estimates. By establishing the accuracy of our processor queueing submodel, we can confidently integrate it into a multiprocessor system model.

7. Conclusion

This paper presents a processor queueing simulation model for a modern complex processor. For system design using processors which aggressively exploit instruction and thread-level parallelism, we observe that the processor's memory subsystem, which includes caches and buffers, requires the most detailed modeling for memory intensive OLTP workloads. The model implements easily, executes quickly and produces reasonably accurate results. Another advantage of this type of model is that we can vary the model's workload parameters without the need of actual traces, which are often difficult to acquire for large commercial workloads. It is useful for analyzing future workloads or complex workloads for which traces are not available.

The paper shows that burstiness or clustering of stores in OLTP workload has a significant impact on performance, and should be included when evaluating the buffer sizes. The model is used to analyze the overlap in miss penalty with increasing memory latency. It is shown that the buffer sizes should be tuned by designers to achieve good overlapping when the latency increases. We have discussed a few methods that we are investigating to integrate the processor model with a system model.

Acknowledgement

We would like to thank Sudarshan Kadambi and Vijay Balakrishnan for providing the cycle accurate simulation results, Robert Lane for reviewing the paper, Robert Cypher for prompting the thought of modeling cache misses overlap, and Anders Landin for insisting to see validation with more stressful workload.

References

[1]D.H. Albonesi and I. Koren, "A Mean Value Analysis Multiprocessor Model Incorporating Superscalar Processors and Latency Tolerating techniques," Int'l Journal of Parallel Programming, 1996.

[2]M. Durbhakula, V. Pai, and S. Adve, "*Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors*," Proc. Intl. Symp. on High Performance Computer Architecture, 1999.

[3]S. S. Mukherjee, et. al., "Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator," Workshop on Performance Analysis and Its Impact on Design, June 1997.

[4]V. Krishnan and J. Torrellas, "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors," PACT'98, October 1998.

[5]S. Pai, P. Ranganathan, and S. Adve, RSIM Reference manual, Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, Aug, 1997.

[6]V. S. Pai, P. Ranganathan, and S. Adve, "The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology," Proc. Intl. Symp. on High Performance Computer Architecture, 1997.

[7]M. Rosenblum, et. al., "Using the SimOS Machine Simulator to Study Complex Computer Systems," ACM Transactions on Modeling and Computer Simulation, 1997.

[8]E. Schnarr and J. Larus, "Fast Out-of-Order Processor Simulation using Memorization," ASPLOS-8, October 1998.

[9] E.Schnarr, M. Hill and J. Larus, "Facile: A Language and Compiler for High-Performance processor Simulators", PLDI'01, 2001.

[10]D. Sorin, V. Pai, S. Adve, M. Vernon, and D. Wood, "Analytical Evaluation of Shared-Memory Systems with ILP Processors," Proc. Intl. Symp. on Computer Architecture, 1998.

[11]Sun Internal Presentation, R. Cypher.

[12]Sun Internal Presentation, N. Kosche and Q. Jacobson.

[13]TPC Benchmark C Standard Specification, Transaction Processing Performance Council.

[14]D. Willick and D. Eager, "An Analytical Model of Multistage Interconnection networks," in Proc. ACM Sigmetrics, May 1990.

[15]M. Vernon, E. Lazowska, and J. Zahorjan, "An Accurate and Efficient performance Analysis Technique for Multiprocessor Snooping Cache Consistency protocols," International Symposium on Computer Architecture, 1988.

Session 4

Workload Characterization

Performance Analysis of Speech Recognition Software

Chunrong Lai, Shih-Lien Lu and Qingwei Zhao
Intel Corporation

Comparison of Memory System Behavior in Java and Non-Java Commercial Workloads

Morris Marden, Shih-Lien Lu, Konrad Lai
and Mikko Lipasti
University of Wisconsin – Madison
Intel Corporation

Characterizing TPC-H on a Clustered Database Engine from the OS Perspective

Yanyong Zhang, Jianyong Zhang, Anand
Sivasubramaniam, Chun Liu and Hubertus Franke
The Pennsylvania State University
IBM T.J. Watson Research Center

Performance Analysis of Speech Recognition Software

Chunrong Lai, Shih-Lien Lu+ and Qingwei Zhao
China Research, Intel Labs.
(chunrong.lai@intel.com, qingwei.zhao@intel.com,
+Microprocessor Research, Intel Labs.
5350 NE Elam Young Parkway, Hillsboro, OR 97124
(shih-lien.l.lu@intel.com)

Abstract

This paper characterizes the behavior of a speaker-independent large vocabulary continuous speech recognition (LVCSR) system. This system is used to dictate Chinese (Mandarin) utterances of different speakers and achieves a word recognition accuracies of 85%~96% depending on the cleanness of input signals and the complexity of the spoken sentences. Several methods are used to characterize its processing behavior. First, the same system is run on different Intel platforms and their performance measured. Second, we use an Intel performance monitoring toolset –Vtune to read hardware counters build in the CPU. These counters measured the instruction distribution as well as processor utilization rate. Third, a full platform simulator SoftSDV together with a cache simulator is used to study its memory behavior in more detail. We find this software system to have a large memory working set. Data access to first level cache has good locality. There are two groups of memory usage displacing each other in the second level cache. Second level cache miss rate declines much fast after the size increases beyond 8MB. We also identify a few instructions that cause a larger number of level-2 cache misses. Using software prefetching we improve the overall performance by an average of 7%.

1. Introduction

Speech recognition (SR) by computer has been an active research for a while now. It provides a natural interface for human to interact with machines. Many speech recognition systems are available in the market such as Dragon Systems' Dragon Naturally Speaking [1], IBM's viaVoice [2], L&H's Voice Xpress [3] and Philips' FreeSpeech [4]. However, general adoption of machine speech recognition is still not widely accepted due to recognition unreliability. Much research efforts continue to be devoted in perfecting speech recognition techniques in recent years. So far the most widely accepted technique is based on hidden Markov Model (HMM). It is used in most state-of-the-art SR systems in the world. These large vocabulary continuous speech recognition systems based on HMM often face the issue of trading-off between the recognition accuracy and the computation speed. Thus it is important to understand the behavior of LVCSR in order to improve its performance.

Unfortunately processor performance characteristics of speech recognition applications are not well published. Agaram et. al. [5] analyzed the characteristics of a public domain speech recognition engine called Sphinx [6]. This engine has a vocabulary of 21000 and is based on semi-continuous hidden Markov Model (SCHMM). Intel Labs, Intel China Research Center (ICRC), has developed a LVCSR engine that was originally licensed from Oregon Graduate Institute (OGI) [7][19]. The vocabulary of this system is over 51000 words and is based on continuous hidden Markov Model (CHMM) with multivariate mixture Gaussian as observation density to cover speech signal variability. With sufficient training data, CHMM systems gives better recognition performance. However, these systems are more complex and slow down recognition speed. ICRC's LVCSR engine has been extensively tested with various Chinese (Mandarin) speakers and can achieve speaker-independent word recognition accuracies of 85%~96%. The variation of accuracy is due to the cleanness of input signals and the complexity of the spoken sentences. In this paper, we examine the behavior of ICRC's LVCSR speech-engine.

We study this engine with several tools. First, we measure the running time of this LVCSR engine on different Intel Architecture based platforms. These platforms vary in CPU frequency and level-2 cache size. We, then, use a performance analysis tool called VTune™ [8] to uncover more detail characteristics of the program. Vtune supports both event and time based sampling. It also allows users to perform call graph profiling. We mainly use the event-based sampling (EBS) capability to collect run-time information by reading the performance monitoring counters on the processor. During event-based sampling, VTune interrupts the processor after a specified number of event occurrences, and collects a sample containing the instruction addresses where the event occurs. At the end of EBS, we determine how many times an event occurred by the collected data. Information collected with VTune includes instruction distribution, branch misprediction rate, cache misses, and instruction/micro-op executed per cycle. These

data are compared with some of the SPEC 2000 CPU benchmark programs [9]. As expected, there are many floating-point computations in this program. However many of them are of the form ADD/SUB followed by Multiply instead of the other way around. Moreover, we observe that around 65% of the time the processor is not retiring any instructions/micro-ops while running this LVCSR program. It also has a relative large second level cache miss rate. We perform further study on the memory behavior of this program using software simulation.

In order to consider all the effects, including system calls, we decide to use a full system simulator. We ran this LVCSR program on this full system simulator called SoftSDV [10], in batch mode using speech input captured in files. Due to long simulation time we generate memory traces and run these traces through a trace-driven cache simulator with different cache configurations. The results show that this engine in particular, and speech recognition applications in general, has large memory working sets. Memory references have good spatial locality and not as good temporal locality. Level 2 (L2) cache's miss ratio changes at a different rate after its size grow larger than 8 MB.

2. General features of LVCSR

A speech recognition system converts speech into text strings. An uttered sentence is digitized first. These digitized samples are grouped in overlapping frames. A set of features [11] capturing the characteristics of a frame is extracted. This set of features is referred to as an observation. Thus, an uttered sentence is represented by a group of observations: $O = o_1, o_2, o_3, ..o_{t-1}, o_t$. The goal of the speech recognition system is to find the most possible word-sequence, $W = w_1, w_2, w_3, ..w_{n-1}, w_n$, that matches the observations. This process is express mathematically [12] as:

$$W = arg\ max_W\ P(W|O) \qquad (1)$$

The right hand side of equation (1) can be re-written due to *Bayes* rule:

$$W = arg\ max_W\ P(W)P(W|O)/P(O) \qquad (2)$$

In equation (2), the probability of observed input sequence *P(O)* is constant. The probability of a word sequence *P(W)* is part of the *language model* and acts as a search constraint. The conditional probability *P(W|O)* measures how well the word string *W* matches the given input *O*. This conditional probability is obtained during the training phase and is usually referred to as the *acoustic model*.

Acoustic models are built for pre-specified acoustic units. In our system, phonemes (phones) are the basic acoustic units. Each phone in the language is modeled by an HMM. An HMM consists of two stochastic processes. One is a *hidden* Markov chain, which accounts for *temporal* variability. The other is an observable process, which accounts for *spectral* variability [13]. These two have proven to be very powerful to cope with most sources of speech ambiguity, and very flexible to allow the realization of recognition systems with dictionaries of tens of thousands of words. Each HMM consists of several states and different phones may share a state. Several phones are grouped to form a word, and words are collected to construct a sentence. Therefore a given sentence can be thought of as a collection of phones. There are thousands of HMM states in our system. Figure 1 illustrates an example HMM, which contains three regular states and two pseudo states (a start and an end).

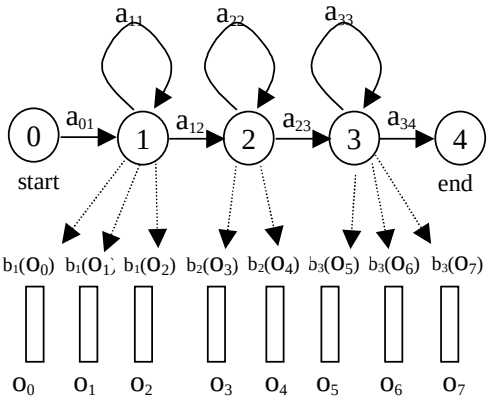


Figure 1. A five-state HMM with two pseudo states

The transition from state *i* to state *j* is determined by the transition probability a_{ij}. At each time unit, a speech observation o_t is matched according to the output density function $b_i(o_t)$. HMMs can be classified according to the nature of the elements of the *B* matrix, which are distribution functions. *Discrete HMMs' (DHMMs)* distributions are defined on finite spaces. In this case, observations are characterized as discrete symbols chosen from a finite alphabet . In detail, each incoming observation vector is replaced by the index of the closet vector in a precomputed codebook, and the output probability functions are just lookup table containing the possibilities of each possible VQ (vector quantization) index. Distributions are associated with model transitions. A transition probability and an output distribution on the symbol set are associated with every transition.

Another possibility is to define distributions as probability densities on continuous observation spaces. In this case, strong restrictions have to be imposed on the functional form of the distributions, in order to have a manageable number of statistical parameters to estimate. The most popular approach is to characterize the model transitions with mixtures of base densities function having a simple parametric form. The base densities function is usually Gaussian or Laplacian. The mean vector and the covariance

matrix can parameterize the base densities function . HMMs with these kinds of distributions are usually referred to as *continuous HMMs (CHMMs)*. In order to model complex distributions in this way a rather large number of base densities has to be used in every mixture. This may require a very large training corpus of data for the estimation of the distribution parameters.

In *semicontinuous HMMs (SCHMMs)*, for example [14], all mixtures are expressed in terms of a common set of base densities. Different mixtures are characterized only by different weights. A common generalization of semicontinuous modeling consists of interpreting the input vector as composed of several components, each of which is associated with a different set of base distributions. The components are assumed to be statistically independent. The distributions associated with model transitions are products of the component density functions. Computation of probabilities with discrete models is faster than with continuous models, nevertheless it is possible to speed up the mixture densities computation by applying vector quantization (VQ) on the Gaussians of the mixtures [15]. Interested readers may find more details on their differences from [16].

As mentioned, in CHMM each HMM state contains large number of parameters of Gaussian distribution. In our system, each state requires 3 ~ 4KB of space to store these parameters. With thousands of state in our system the amount of storage required to store these parameters ranges in tens of million bytes (MB). Each time an input frame (represented as features) comes, Gaussian computation is processed using these parameters to get a probability of how close the observation (frame) matches a state is. Pronunciation dictionary is organized as a lexicon tree as illustrated by Figure 2 for searching. When the searching process reaches the leaf nodes of a lexicon tree, known as *word boundary*, the language models are accessed to get a score in addition to the acoustic score. Thus the recognition computation involves the calculation of the likelihood function $P(O|\lambda)$, where λ is a HMM. The recognition process consists of searching through all possible phones sequence, by looking up the acoustic model and the language model, and then finds the most meaningful (or with the highest score) sequence. In order to reduce the amount of possible branches needs to be searched, many unlikely paths are pruned during the search process.

The storage requirement of the language model is determined by the size of the vocabulary and is usually larger than the acoustic model. With a vocabulary of more than 51,000 words, the size of the language model and language look-ahead model is around 110MB. Besides the acoustic model and language model, there is an additional storage space requirement for the recognition process. During the search process, space is needed to construct lexicon tree and all current likely paths under consideration. This is named the search buffer and its size ranges in tens of MB also. When the search process arrives at the word boundary, accesses to the search buffer and the language model interact with each other. During the normal search mode, accesses to the search buffer and the acoustic model are interleaved. All these three spaces are large in size and may displace each other from cache memory.

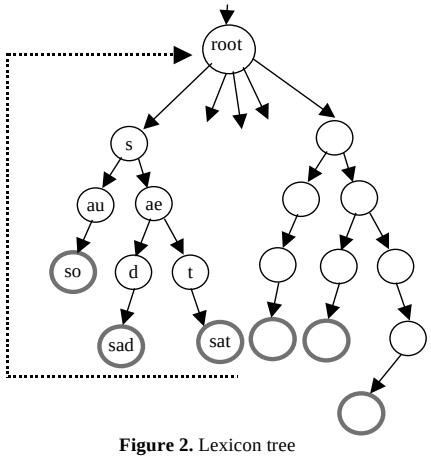


Figure 2. Lexicon tree

The technique named language model look-ahead is used [17][18] to reduce possible search space, thus reduces the computation time and search buffer space. This technique adds language model information into the node of a lexicon tree and further prunes the tree according to it. With this look-ahead technique fewer searches can reach the word boundary [19]. It increases the access frequency of the language model. However, it is still less than the access frequency of the acoustic model. The search buffer maintains dynamic data structures used to store all possible recognition results. These structures are very input dependent. For example, with noisy speech signals the size of search buffer may be tens of the size of clean speech signals.

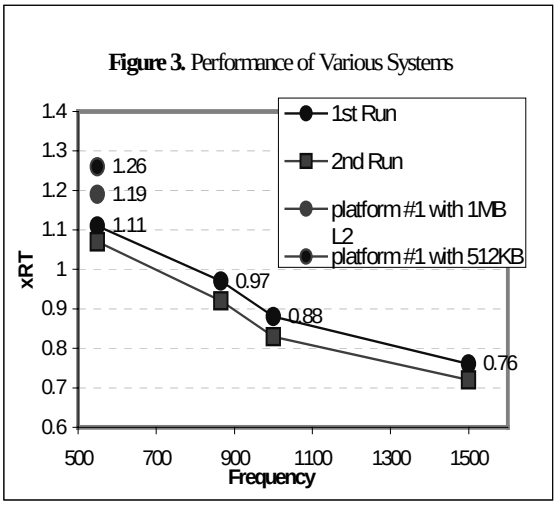
During the search, beam values can be used to help decide if a search path should continue. If a score is not much lower than the current highest score according to the beam value, the path is likely to survive. We will keep the path in the search buffer. Thus the larger the beam value, the more space is needed to keep all temporary data around. More computations and search effort are required also. Matching (Gaussian computation) of continuous HMM to find the most likely phone spoken requires a lot of computation. Moreover, maintaining of the uncertain number of tokens in such a big tree also needs to be handled carefully. Good pruning algorithm can reduce the number of active nodes, active states and active tokens dramatically. For example, in our current system the active number of states is generally less than 1% of the total number. Many other optimization techniques at the implementation level have been applied to the speech

recognition system. For example, the pointer-base lexicon tree can be reordered based on some chunk with better data locality. Prefetch instructions can be employed inside the Gaussian mixture computation of the large HMM state since the data locality is better there. Also with the access locality, the language model look-ahead [7][18] probability of a node can be computed speculatively and buffered with the computing of its brother nodes. We have mentioned that different nodes may share a same trained state. A buffer is used for remembering the results of these Gaussian mixture computations. Thus, we avoid duplicating these Gaussian mixture computations if the probability has already existed in the buffer. Also because of the continuity of the voice, when a frame is computed with a HMM state, the following several frames can also be computed with the same state speculatively so as to utilize the state data more efficiently to avoid future stalls. This buffer becomes the fourth memory working set of the system. Now in the processing of normal nodes, the buffer interleaved with the search space, only when the buffer misses, the HMM states of the acoustic model are accessed. All these optimizations and speculative computations increase the data locality of the system and speed up the system. We also use tuned performance library [20][21] with Streaming SIMD Extension (SSE) instructions as much as possible to speed up the application.

3. Analysis Results

As mentioned, we analyze the speech recognition software using three approaches. First, we run the application on different platforms to understand how well the application scale with processor advancement. Speech input is captured in files and ran in batch mode in order to control the quality of the samples. We then use an Intel toolset called Vtune to collect built-in performance counter values [22][23]. Finally we use a full system simulator called SoftSDV together with a cache simulator to study the memory behavior in more depth.

Figure 3. Performance of Various Systems



3.1 Real time performance on different platforms

We use the indicator xRT (real time ratio) to measure the speed of the speech engine. An xRT of 1 means the time to decode is the same as the speech signal time. A lower value of xRT means better performance. We run the LVCSR engine on the following four platforms.

1. Pentium®III 550 MHz with 2MB off-chip L2 cache, 512MB SDRAM. (With 440GX AGPset)
2. Pentium III 866 MHz with 256KB L2 on-chip cache, 512MB RDRAM. (With 840 Chipset)
3. Pentium III1000 MHz with 256KB L2 on-chip cache, 512MB RDRAM. (With 840 Chipset)
4. P41.5GMhz with 256KB L2 on-chip cache, 512MB RDRAM. (With 850 Chipset)

First, each machine is restarted and then the speech engine is run. During this 1st run cache and TLB are all just initialized. After the first run, this LVCSR engine is terminated and is started again. We again capture the running time the 2nd time. Figure 3 illustrates the results of these two runs on the above listed four platforms. The 2nd and the 3rd platforms are identical except the CPU frequency. The 1st platform uses a slower CPU, however the level 2 cache is off-chip and larger in size. The speed of the off-chip cache is half of the core speed, while the on-chip cache has the same speed as the core. Moreover, the 1st platform uses SDRAM instead of RDRAM thus it has a lower memory bandwidth. Two more data points were collected for the 1st platform by switching the CPU with different L2 sizes. It seems a larger L2 even with a slower speed tends to compensate the lack of CPU frequency.

For the 1st platform we further experiment with the memory requirement by varying the amount of the SDRAM in the system and collected the real time rate and CPU utilization rate. Figures 4 and 5 summarize the results. We see both the real time performance and CPU utilization level off after 256MB. Unfortunately we are unable to further delineate between 256MB and 128MB. Having a main memory of 256MB is essential for running this application.

3.2 Computation distribution

Both PIII/P4 processors decode instructions to µ-ops run internally. Figure 6 summarizes the instruction per cycle (IPC) and the micro-ops per cycles (UPC) of different platforms obtained with Vtune™. Even though the 2nd and the 3rd platforms differ only in CPU frequency, their IPC and UPC are different due to memory access.

The rest of this section's analysis is based on the 2nd platform. The computation distribution of running this LVCSR obtained from Vtune™ is shown in

Figure 7 and 8 separated by major tasks in the application. Figure 7 illustrates the execution time distribution among four major tasks while Figure 8 shows the MIPs distribution. Both graphs shows that majority of the time and MIPs is spend on Guassian computation and search.

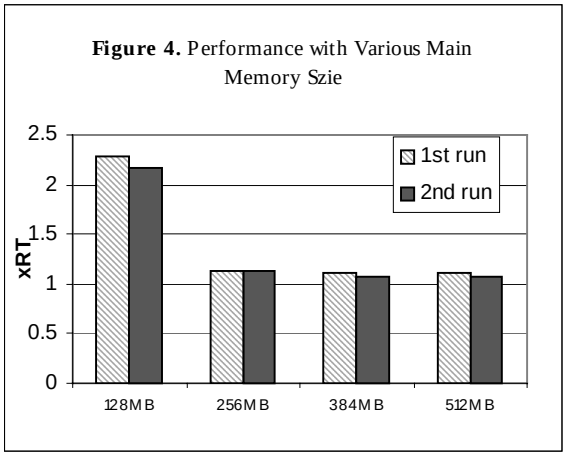


Figure 4. Performance with Various Main Memory Szie

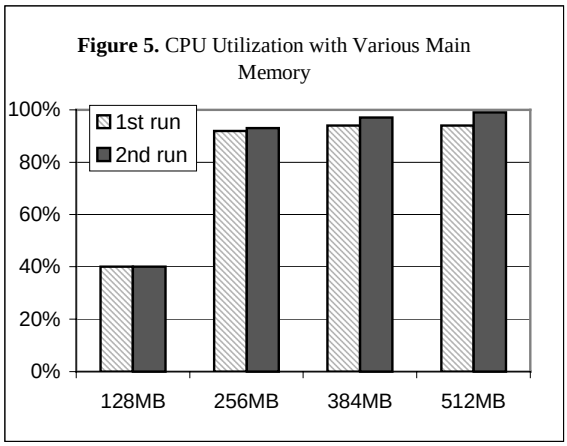


Figure 5. CPU Utilization with Various Main Memory

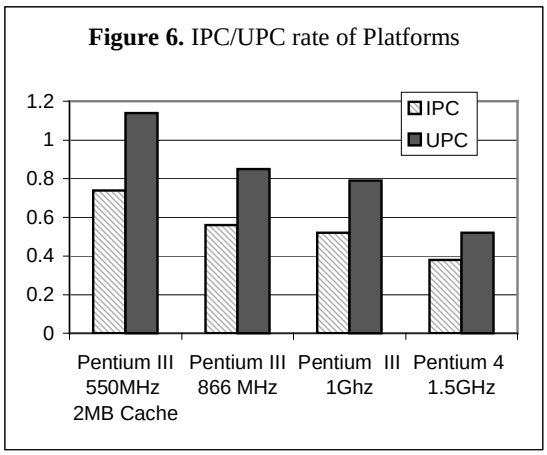


Figure 6. IPC/UPC rate of Platforms

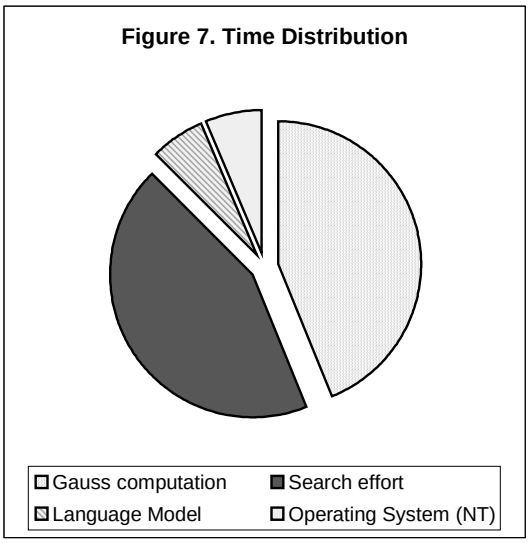


Figure 7. Time Distribution

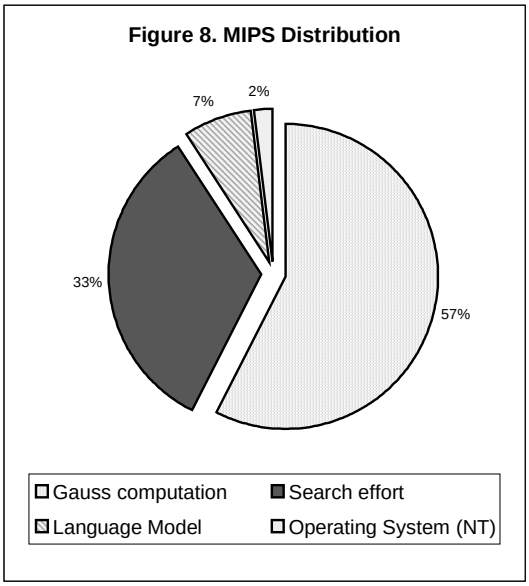


Figure 8. MIPS Distribution

The most frequently seen computation patterns observed are *((a-b)^2)*c* and *log(1+exp(x))*. The later is approximated with a polynomial equation. Both patterns can be expressed with ADD/SUB then Multiply instruction sequences. This is different from the Multiply-ADD/SUB sequences seen in DSP type applications. Usually during the search a great percentage of the actual search time is spend by the Gaussian computation. According to previously reported analysis, Gaussian computation may stall the search process due to dependency. Work done by [17] reports that typically 40%~80% of the total execution time is due to Gaussian Computation. More recently, studies have shown that it is not less than 75% [18]. Our engine, without modification, also reports more than 70% of execution time on Guassian computation. An optimization is used at the algorithm level that speculates the result of computation and reuses results from previously computation. With this optimization

technique we are able to reduce this ratio greatly. It is worthwhile to note, since the cost of the system idle process is counted as OS activity, the actual cost of the language models (include the language look-ahead model) should be higher than the value reported.

When we compare the time distribution with the MIPS distribution we found some small discrepancy. This is because the Gaussian module is composed of more regular and more inherently parallel computations while the search effort is more random. So the time distribution ratio of it is less than the MIPS distribution ratio. The search effort module, also called token propagation module, has many branch and data dependency. Thus, the time distribution ratio of the search module is larger than the MIPS distribution ratio. Unfortunately, these two modules are somewhat interleaved making parallelizing the Guassian module challenging.

There are also some counters for the specific type of instructions. Though they are not detailed enough, some understanding can be obtained. Table 2 lists the break down of instruction types and corresponding miss rates. We will discuss the comparison between LVCSR and other SPEC benchmark programs later.

3.3 Where have all those cycles gone?

We further investigate where have the time been spend in the program. Again all data are collected on the 2nd platform using VTune. Figures 9 and 10 show that about two thirds of the execution cycles are idling and retiring no instructions. This indicates there may be many dependency chains in the application. Note that the maximum number of u-ops can retire each cycle is 3. Since one instruction may be decoded into multiple u-ops, the number of cycles where 3 instructions are retired is less than the number of cycles where 3 u-ops are retired.

Figure 9. Breakdown of IPC

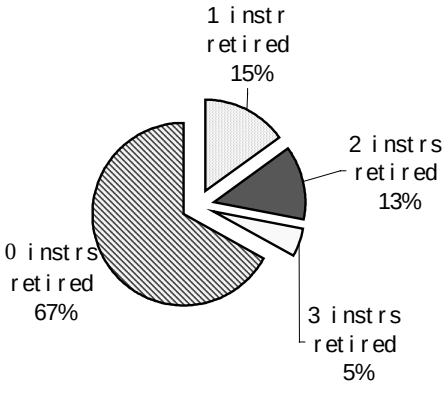
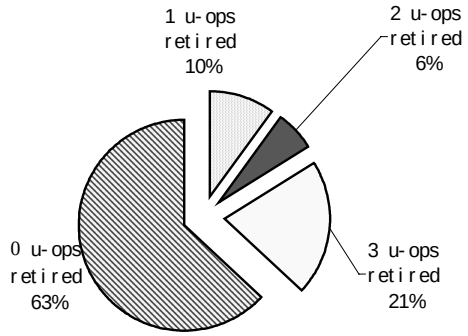


Figure 10. Breakdown of UPC



There are additional counters in Pentium III allowing us to further categorize different types of stalls. Table 1 summarizes the result. The percentage is ranked among all cycles. Accounting for all cycles is difficult in an out-of-order processor such as Pentium III. Execution can proceed during a stall cycle in some other part of the machine. For example, when the pipeline has a resource stall, instruction can continue to be fetched. Similarly, during is a instruction fetch stall, there are other instructions being executed in the other part of the pipeline.

Resource related stalled cycles	52%
Instruction fetch stalled cycles	4.5%
Partial stalled cycles	1.5%

Table 1. Stall cycles

Instruction cache misses and ITLB misses cause instruction fetch stalls mainly. Resource-related stalls are much higher than instruction fetch stalls and partial stalls in LVCSR. This is similar to floating point benchmarks. The resource-related stalls indicate that there are instructions in the application requiring the same hardware resources such as register renaming buffer entries and memory buffer entries [22]. Branch misprediction recovery and delay in retiring mispredicted branches also causes resource-related stalls. Since branch misprediction rate and L2 cache miss rate are not overly high, long dependency chains in the code may have caused these stalls cycles.

At the mean time, there is a large percentage (21%) of the cycles that can retire 3 μ-ops. This indicates that the there are parallelism computations in the program.

3.4 Comparison with SPEC CPU2000 benchmarks

Using the same counters accessible through Vtune™, we collect data for some SPEC CPU2000 benchmarks. Table 2 summarizes the comparison. These SPEC 2000 benchmarks are compiled with Microsoft Visual Studio compiler using default optimization options (-Ox -Zi). Some numbers are not included if they are small and negligible. As mentioned before LVCSR does have large memory requirement and there may have long dependency chains in the program that need data from critical loads. We see the program characteristics of LVCSR speech engine are similar to a typical floating-point application in the SPEC2000[9].

			SPEC 2000 Integer						SPEC 2000 FP		
		lvcsr	gcc	Gzip	eon	parser	perlbmk	vpr	mesa	art	ammp
Execution Aggregate	IPC	0.56	0.41	0.99	0.71	0.57	0.94	0.45	0.64	0.18	0.41
	UPC	0.85	0.75	1.21	1.56	0.86	1.36	0.65	1.12	0.23	0.58
Stall cycles	Resource related	52%	44%	27%	16%	44%	15%	60%	23%	88%	74%
	Partial Stall	1.5%	1.1%		1.9%	11%	3.2%	1.0%	8.4%		
	Instruction Fetch	4.5%	8.5%	1.3%	9.0%		10%	1.6%	1.6%		
Instruction Mixes	Branches	10%	22%	19%	14%	21%	21%	17%	11%	14%	9.6%
	FP	8.5%						5.2%	8.4%	12%	33%
	SSE	16%									
Memory reference/instr.		0.60	0.65	0.40	0.83	0.62	0.51	0.70	0.68	0.58	0.58
Branch mis-prediction		6.4%	7.4%	5.3%	6.1%	5.3%	4.6%	7.9%	2.8%	5.2%	2.3%
L2 instruction fetch/hundred instr.		0.26	0.2	0.03	0.20	0.06	1.6	0.26	0.27	0.12	0.06
Cache miss	DL1	2.8%	4.3%	7.2%	0.14%	2.8%	1.2%	3.8%	0.78%	15%	4.5%
	L2	38%	40%	5.4%	0.27%	35%	6.8%	38%	13%	96%	52%

Table 2. Comparison between LVCSR and SPEC CPU2000 benchmarks

3.5 More detail memory behavior study

As mentioned, besides using Vtune™ we also use software simulation tools to further study the memory behavior. With a built-in first level cache in the SoftSDV's CPU model, we first skip 66.5 billion instructions after starting up SoftSDV. We use 1 billion instructions for warming up cache then collected a 380 million L2 references trace. These traces are used as input to a trace-driven cache simulator. When simulating using this trace we use the first 200 million references to warm the L2 cache.

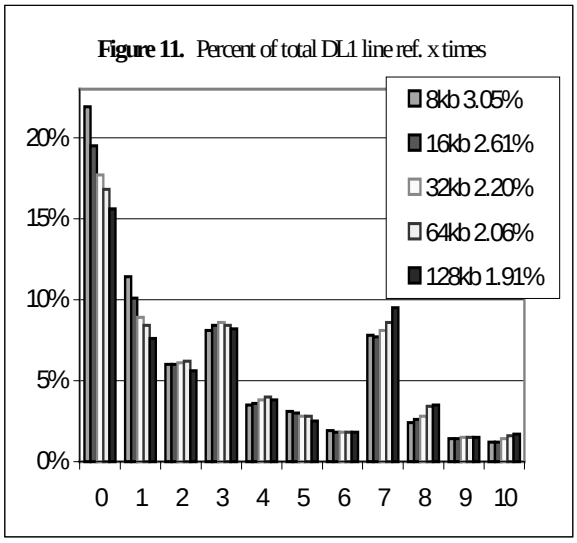


Figure 11. Percent of total DL1 line ref. x times

We first analyze the level 1 cache behavior. The DL1 has relatively good locality. Its miss rate is comparable to "ammp" in the SPEC2000 FP benchmarks. Figure 11 illustrates the cache line reference frequency for various DL1 cache sizes. In this figure the x-axis is the number of times a cache line is referenced before it is replaced. The y-axis is the percentage of lines among all referenced lines. We also show the miss rate for each DL1 sizes in the legend. This behavior is very similar to "ammp". However the L2 cache miss rate of LVCSR is higher than "ammp". There are other SPEC2000 FP benchmarks which act very differently from LVCSR and "ammp".

Figure 12 plots the miss rate verses L2 cache size for various line sizes. Note this is a log/log plot. We see a change of miss rate slope after 8MB. The size requirement is somewhat expected because we know there are three large data storage spaces used for acoustic models, language model and search buffer. Another fact is that our data structure is somewhat large. Each state has 3 to 4 KB of Gaussian parameters. During the search each state is visited one by one. Therefore it is more advantage to have large line sizes. Figure 4 also shows that every time L2 line size is doubled we get similar miss rate as having twice the cache size.

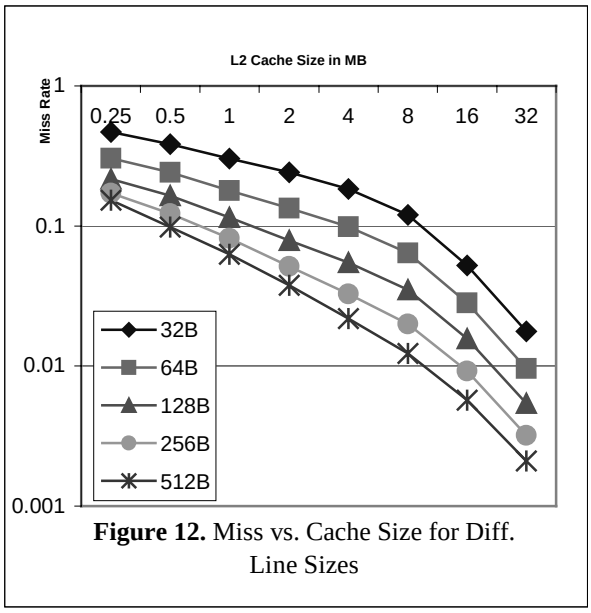
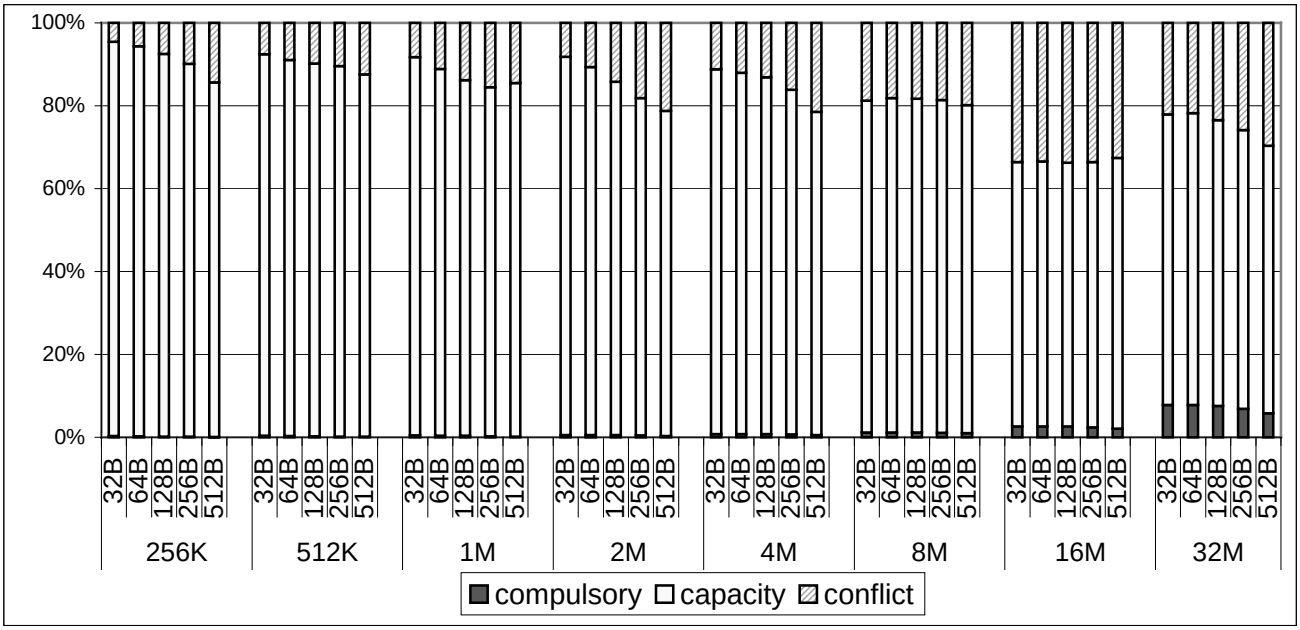


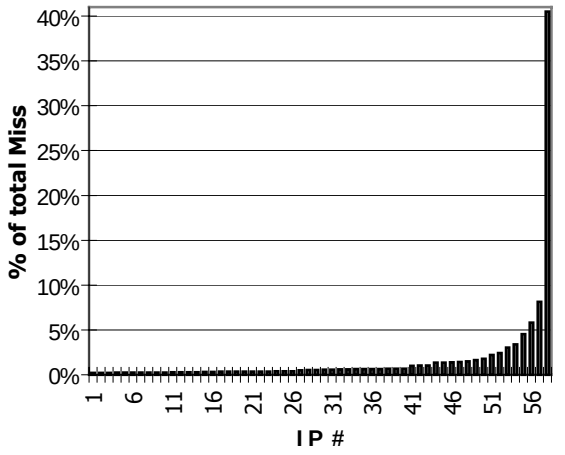
Figure 12. Miss vs. Cache Size for Diff. Line Sizes

Figure 13 shows the capacity ratios of the cache miss with various L2 sizes and L2 line sizes. These data are collected with very long traces. We first skip the first 66.5 billion instructions after starting up SoftSDV. A L1 data cache is built-in with SoftSDV. We use 1 billion instructions to warm up the L1 data cache first. L1 data cache is a typical 32KB 4-way set-associative cache with line size equals to 32B. We then collect around 400 million references to L2. We use about half of these traces to warm up and start collecting statistic after the warm up period. Of traces used to collect statistic 74% are reads. When L2 cache size is small, most of the misses are capacity misses. Very little conflict misses are observed. That may indicate cache associativity need not to be high if we have smaller cache for LVCSR. The L2 capacity miss ratio shows a small drop after the size reaches 16MB.



When we collected traces, we associated with each data reference in the trace the corresponding instruction pointer (IP) address. When we sort these IP addresses we found about 58 IPs that cause 90% of the L2 misses. Moreover, one top IP address causes more than 40% misses (more than 36% of all non-compulsory misses). Figure 14 shows the distribution and accumulated miss rate for these 58 IP addresses. By inserting software prefetch for this one instruction we are able to improve the overall application performance.

Figure 14. Miss percentage vs. Top 58 IPs



Using traces for cache study is faster but have limitations. In the future we will interface the full system simulator with a cycle by cycle timing accurate memory subsystem model. With that we will be able to collect performance directly instead of estimating the performance impact from miss rates.

4. Conclusion

LVCSR systems based on continuous HMM are gaining usage. We use several tools available to us to study the general characteristic of a LVCSR developed by ICRC. Our study indicates that this LVCSR requires a large number of floating-point computations to evaluate GMM. Since a HMM state uses a large data structure (which typically is 3KB~4KB), there is good data spatial locality. However accesses to language model and acoustic model are random and thus exhibit little temporal locality. This may be one of the main causes of stalled cycles. We also use software simulation to narrow down on instructions that cause most of the level 2 cache misses and use software prefetching to improve performance. Further memory behavior study on the long trace collected will be performed.

Acknowledgement

We thank Konrad Lai, Hubert Hum and John Shen for their discussion and comments. Thanks also to the anonymous reviewers for their many helpful suggestions.

4. References

[1] http://www.dragonsys.com

[2] http://www-4.ibm.com/software/speech

[3] http://www.lhs.com/voicexpress

[4] http://www.speech.be.philips.comhttp://simos.stanford.edu/

[5] K.Agaram, S.W.Keckler, and D.C.Burger. "A Characterization of Speech Recognition on Modern Computer Systems", *4th IEEE Workshop on Workload Characterization*, at MICRO-34, December, 2001 , 2001.

[6] http://fife.speech.cs.cmu.edu/sphinx

[7] Qingwei Zhao, Zhiwei Lin, Baosheng Yuan and Yonghong Yan, "Improvements in search algorithm for large vocabulary continuous speech recognition", ICSLP, Vol.4, October, 2000, Beijing, pp306-309.

[8] http://developer.intel.com/software/products/vtune/index.htm

[9] SPEC CPU2000, http://www.specbench.org/osg/cpu2000

[10] R. Uhlig et. al., "SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture," Intel Technology Journal, 4th quarter, 1999. http://developer.intel.com/technology/itj/q41999/articles/art_2.htm

[11] R. Haeb-Umbach, D. Geller, and H. Ney. Improvements in connected digit recognition using linear discriminant analysis and mixture densities. In ICASSP, pages 239--242.

[12] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," Readings in Speech Recognition, Kaufmann, 1990, pp. 267-296.

[13] S.Young A review of large-vocabulary continuous speech recognition. IEEE Signal Magazine, 1996, Sep, 45-57

[14] X. D. Huang, Y. Ariki, and M. Jack. *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, 1990.

[15] E. L. Bocchieri. Vector quantization for the efficient computation of continuous density likelihoods. In ICASSP, pages 692--694.

[16] H. W. Huang, M. Y. Hon, M. Y. Hwang and K. F. Lee, "A Compartive Study of Discrete, Semicontinous, and Continuous Hidden Markov Models," Computer Speech and Language, 1993, No. 7, pp. 359-368.

[17] P Beyerlein, et. al., "Hamming distance approximation for a fast log-likelihood computation for mixture densities". Proc. of the European Conf. on Speech Communication and Technology, Madrid, Spain, vol. II, pp.1083-1086.

[18] S. Ortmanns, et. al., "Language-model look-ahead for large vocabulary speech recognition", ICSLP, 1996, pp2095-2098.

[19] Y.Yan, X. Wu, J. Schalkwyk, R. Cole. "Development of CSLU LVCSR: The 1997 D ARPA HUB4 Evaluation System". In Proceedings *DARPA '98 BNTUW*, 1998

[20] Intel Corporation, Intel Integrated Performance Primitives, http://developer.intel.com/software/products/ipp

[21] Intel Corporation, Intel IA-32 architecture software developer's manual, http://developer.intel.com/design/pentium4/manuals

[22] D. Bhandarkar and J. Ding, "Performance Characterization of Pentium® Pro Processor," Proc. of Symp. On High Performance Computer Architecture, Feb 1-5, 1997, San Antonio pages

[23] Yong Luo, Kirk W. Cameron, Josep Torrellas, and Yan Solihin, "Performance modeling using Hardware Performance Counters", Tutorial, HPCA-6, Jan 2000, Toulouse, France.

Comparison of Memory System Behavior in Java and Non-Java Commercial Workloads

Morris Marden[1], Shih-Lien Lu[1], Konrad Lai[1], Mikko Lipasti[2]

[1]Microprocessor Research, Intel Labs
Intel Corporation
Hillsboro, OR 97124
{morris.marden, shih-lien.lu, konrad.lai}@intel.com

[2]Dept. of Electrical and Computer Engineering
University of Wisconsin
Madison, WI 53706
mikko@ece.wisc.edu

Abstract

In this paper, we compare the memory system behavior of Perl and Java versions of SPECweb99. We find that the memory behaviors of these two versions of SPECweb99 are very different. The Java version incurs 213% more cache misses than the Perl version for a 1 MB second level cache and 179% more misses for a 16 MB cache. Much of this increase is due to true sharing, which is 222% to 415% higher in the Java workload than in Perl. We find that for 1 MB caches, the exclusive state is effective, since it is able to remove 24% and 61% of the write upgrades for Perl and Java respectively. However, for 8 MB caches, the exclusive state only removes 1% to 5% of the upgrades. Also, it is important to have an efficient mechanism for cache to cache transfers, since 71% to 87% of the second level cache misses hit a modified line in another processor's cache when using 8 MB second level caches. Our early results show that processor consistency can improve the performance of Perl and Java SPECweb99 by 14% and 22% over sequential consistency. Surprisingly, we find that removing all of the serializations in processor consistency degrades the performance of the Java workload by 11%.

1. Introduction

With the growth of the Internet and the World-Wide Web, there has been a dramatic increase in the number of types and styles of programs that computers are being required to run. Traditionally, software engineers wrote programs in high-level languages, like C and C++, which then were compiled into assembly language long before users ran the programs. Later, scripting languages were developed, such as Perl, which interpret each line of code of a program at runtime. Recently, dynamic runtime compiled languages have gained in popularity. These languages, including Java and .Net, are compiled into an intermediate machine language, which is independent of all machines that it is run on. When the user runs these programs, the Virtual Machine compiles these programs on demand for the machine that they run on. Each of these program styles are common today and place different demands on the hardware of the computer. Thus, today's computers must be versatile and perform well with very different program behaviors.

While Java is now a few years old, its memory system behavior is still somewhat a mystery. Recently, there have been a number of papers [2, 4, 9-12, 16, 18-19] that have studied this behavior. However, few studies have compared the behavior of Java workloads to other workloads. Luo and John [12] compared VolanoMark, a Java chat room server, and SPECjbb2000, a java server-side business benchmark that models a warehouse system, to SPECint2000, a general processor benchmark. Cain et al. [2] compared a Java implementation of TPC-W, an online store benchmark, and SPECjbb2000, to SPECweb99, an ISP web server benchmark, and SPECint95. Lepak et al. [10] study the amount of silent stores in a Java implementation of TPC-W, SPECjbb2000, TPC-B (an online transactions database workload), and the SPLASH-2 benchmark suite (a suite of scientific workloads). In these three studies, the Java and non-Java workloads are very different, so one cannot compare the behaviors of the two sets of workloads. In this paper, we compare the memory system behavior of SPECweb99, which uses Perl, to a version that we ported to Java. In addition, of the earlier works of Java workloads, only Cain et al. [2] and Lepak et al. [10] studied the impact of multiprocessor behavior on memory systems. We study the memory system behavior of a four processor machine in this paper.

Even though there have been a large number of studies on memory consistency models, few studies measure their performance differences. The studies that measure performance include [1, 6, 13-14, 17]. Even fewer studies measure performance of consistency models for commercial workloads. Of the above papers, only Martin et al. [13] studied this. In this paper, we include early results of the performance of simple implementations of sequential and processor consistency.

2. Methodology

To compare the memory behavior of Java and non-Java workloads, we used the SPECweb99 benchmark [20], which models a server for the home-pages of users of an ISP. SPECweb99 includes requests for static and dynamic web pages, keep alive and persistent connections, and user targeted rotating advertisements that use cookies. We ported this benchmark from a Perl CGI to a Java Servlet. The Java Servlet performs the same work as the Perl CGI, except that we use shared memory based locks instead of file-based locks in the Servlet. Shared memory locks are more efficient than file locks, since they do not require intervention by the operating system, file system, and disks. However, Perl scripts do not have the ability to use shared memory between processes, so the original Perl implementation is forced to use file-based locks. We believe that this distinction is important, since shared memory locks can have a great impact on memory behavior. Table 1 lists the software that we used for the workload.

To measure the behavior of the memory system, we use the Simics full system simulator [21]. Simics is a functional simulator that simulates multiprocessor systems, using unmodified binary programs. To measure memory behavior, we wrote a memory system model that simulates a two level cache hierarchy and a cycle-accurate multiprocessor split-transaction bus. The bus protocols in our memory model are based on the Pentium II MESI protocol [7] and are tuned for characteristics of processors a few years in the future. Simics sends each memory request to our memory model, which analyzes the effects of the requests and sends the timing information back to Simics. To prevent our results from being skewed, the memory model detects and removes instruction and data read accesses in idle and spin loops. Our memory model also classifies the causes of cache misses and uses Dubois' definitions of sharing [5] to compute the number of true and false sharing misses.

Table 1 shows the configurations that we used for the simulated system. In our experiments, we varied the second level cache line size and second level cache size. The first level cache used the same sized lines as the second level cache. The memory model maintains inclusion between the first level caches and the second level cache, and maintains exclusion between the first level caches (so that self modified code will be handled correctly). We also measured the differences between simple implementations of sequential and processor consistency. Under sequential consistency, the processor can only execute new instructions when the node has completed all outstanding memory requests. However, under processor consistency, the processor does not need to stall for cache misses on writes unless there is a cache miss to the line for a read. Our implementations of these consistency models are discussed in more detail in Section 4.

Operating System	Red Hat Linux 6.2
Web Server:	Apache 1.3.20
Perl:	Perl 5.005_03
Java Server:	Apache JServ 1.1.2
Java:	Sun Java SDK 1.4.0 β3
Processor:	Intel Pentium II
# of processors:	4
Ratio processor to bus frequency:	10:1
Main memory:	1 GB
Minimum memory access latency:	120 processor cycles
Consistency model:	Sequential, Processor
L2 cache read/write ports (total):	1
L2 cache associativity:	4
L2 cache line size:	64 B, 128 B, 512 B
L2 cache size:	1 MB, 8 MB, 16 MB
L2 cache latency:	6 processor cycles
L1 D-cache read ports:	4
L1 D-cache write ports:	2
L1 cache associativity:	1
L1 cache line size:	*same as L2 cache*
L1 cache size (I & D, each):	128 KB
L1 cache latency:	0 cycles (i.e., scheduled in pipeline)

Table 1: Simulated server configuration

SPECweb99 is composed of two parts: a client and a server. The client emulates user requests for static and dynamic web pages from the web server. The server handles these requests and sends responses back to the client. Since we are only interested in the behavior of the server (the system under test or SUT), we simulated the client and server on separate machines and used Simics' network capability to simulate a network between the two simulated machines. We then collected measurements for the behavior of the server machine alone. Figure 1 shows a diagram of the simulated machines.

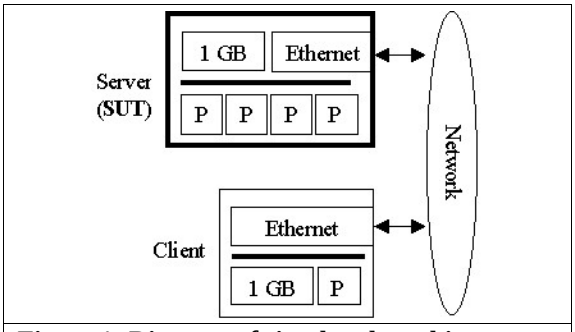


Figure 1: Diagram of simulated machines setup

To obtain reliable steady state results, we first warmed up each of the workloads for a few minutes of simulated time without our memory model (it would be too slow to do this with the memory model). This ensures that our simulator will measure the steady state behavior of the workload. We tuned the load of the workloads to maximize the utilization of the processors. If the load is too high, then the system will spend too much time handling disk accesses and the processors will then be idle while waiting for the accesses to complete. On the other hand, if the load is too low, then there will not be enough work for the processors. In our simulations, we set load of Java SPECweb99 to be the maximum that the client could produce, which was 33% higher than the optimal load of the Perl version.

After warming up the workloads without the memory module, we then ran the workloads with the memory module to collect results. After the first simulated processor (of the server) has executed 250 million instructions, the memory model cleared all of the measurements collected so far. This prevents our results from being skewed due to initial behavior of the simulated system. For example, at the beginning of the simulation, the simulated caches are completely empty, so most of the misses will be compulsory misses. However, when the workload reaches steady state, there will be few compulsory misses. After resetting the measurements, the memory model then measures the memory behavior of the server until the first processor has executed another one billion instructions.

3. Sequential Consistency Results

3.1. Miss Rate

Figure 2 shows the miss rates of the second level caches for the workloads and why these misses occur. Our results show that our Java version of SPECweb99 has a higher miss rate than the Perl version. The Java implementation has a larger working set than the Perl, since the Java version suffers more capacity and conflict misses, especially with relatively small second level caches. The Java Servlet also has much more true sharing misses than the Perl CGI, much of which is due to the shared memory locks. In addition, Java SPECweb99 also incurs much more false sharing misses than Perl, especially at large line sizes.

The sharing behavior of the workloads varies greatly with cache configuration. In relatively small

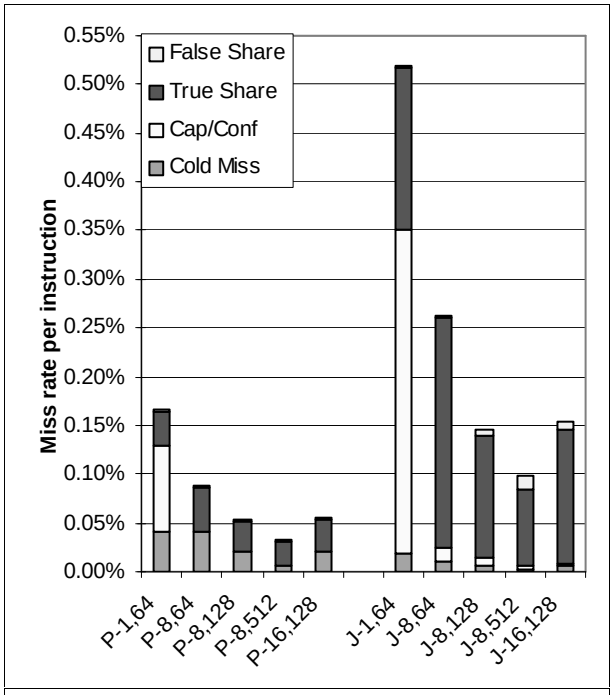


Figure 2: Cache miss rates
Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

second level caches, the large number of capacity and conflict misses reduce the number of sharing misses, since it is less likely that a cache line will be in a remote cache when a processor wishes to write to the line. The number of true sharing misses greatly reduces with line size, especially for the Java implementation. We believe that this occurs since reads and writes are clustered in adjacent cache lines (due to spatial locality). Therefore, when two or more processors share data, a processor will invalidate a large number of consecutive lines in other processors' caches when it writes the data. Later, when the other processors access this data, the large cache lines prefetch more of this data on each access, thereby reducing the number of cache misses. In addition, this reduction in true sharing misses is larger than the increase in false sharing misses, indicating that 512 byte line sizes are useful. Our results also show that increasing the line size is more effective in reducing the miss rate than increasing the size of the cache. In the case of the Java workload, the miss rate actually becomes worse when one increases the size of the cache, since it is more likely that shared data will be in a remote cache.

3.2. Remote Cache Hits

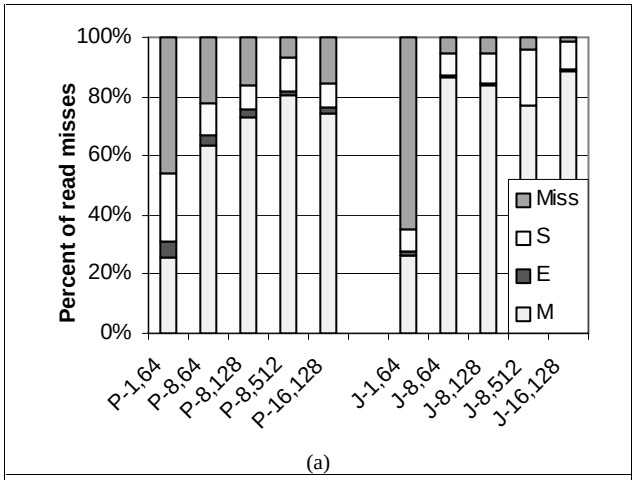
To better understand the sharing behavior of the caches, Figures 3a shows the number of cache misses for data reads that hit in one or more of the other processors' caches. In the Perl version, we see that more than half of the reads hit in a remote cache. For relatively large caches, most of the remote read hits are to lines in the modified state, indicating that it is important to implement an efficient cache to cache transfer mechanism for modified lines. The Java port has an even larger number of remote read hits to the modified state, in part due to the shared memory lock.

The behavior of write misses is similar to that of read misses, as shown in Figure 3b. In the Perl version, there are more write misses that hit in remote modified lines than in the case of read misses. However, there are much fewer write misses that hit in the shared state in a remote cache than for read misses. In contrast, in the Java version, the differences in the number of write misses and read misses that hit in remote modified and shared lines varies based on the line size. One reason that the Java version has more write misses that hit in remote caches in the shared state is that under a highly contented shared memory lock, each processor frequently reads the value of the lock to see if it can acquire the lock. Therefore, the lock will often be in the shared state in a processor's cache when another processor writes to the lock to acquire or release the lock.

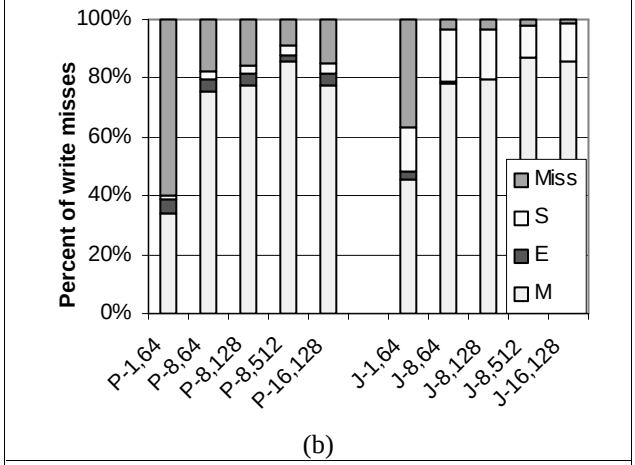
3.3. Exclusive State in Shared Memory Multiprocessors

The exclusive state of the MESI protocol is designed to improve cache performance reducing the number of invalidations that processors use for write upgrades. Recent studies have debated the effectiveness of the exclusive state. Some studies have argued that the exclusive state is not useful, since they found that there are few writes that hit in the exclusive state [3, 8]. However, we believe that a better measure for deciding the usefulness of the exclusive state is the number of lines that enter the cache in the exclusive state and the number of times that the processors can perform silent upgrades instead of invalidations to go from shared to modified. Most writes hit in cache lines in the modified state due to spatial and temporal locality. The exclusive state is not aimed at improving the performance of writes that hit in the cache in the modified state. Using this method, Cain et al. [2] found that the exclusive state is useful for reducing the number of invalidations for write upgrades.

Figure 4 shows that most lines enter the cache in shared state on read misses. The number of shared lines increase with cache size and line size, since it is



(a)



(b)

Figure 3: Misses that hit in remote caches
(a) Read misses
(b) Write misses
Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

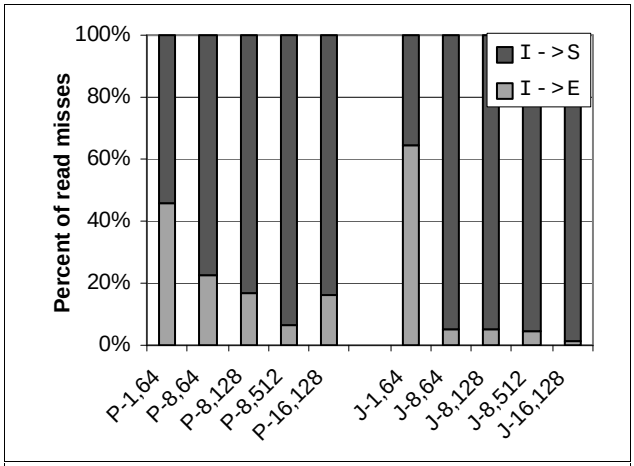


Figure 4: State that lines for read misses enter
Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

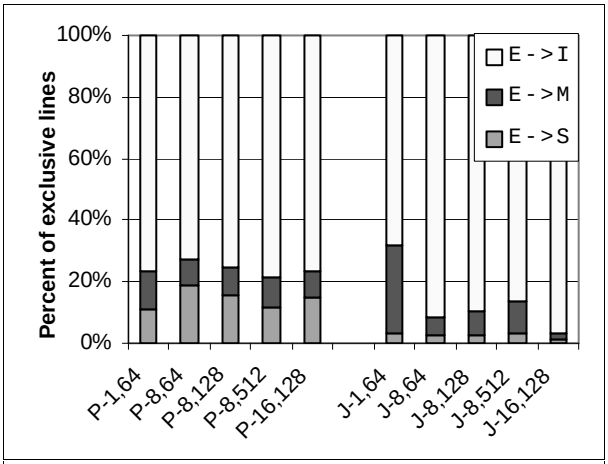


Figure 5: Transitions leaving the exclusive state

Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

more likely that the line will be present in another processor's cache. Here, we see that for relatively large caches, there are more exclusive lines in the Perl version than Java. We also see that the exclusive state is more sensitive to cache line size in the Perl implementation than Java. However, for small caches, the exclusive state is used more in the Java Servlet than the Perl CGI.

In Figure 5, we see that most lines that are in the exclusive state never change to the modified or shared states. In the Perl implementation of SPECweb99, we see that more exclusive lines are downgraded to shared than those that are upgraded to modified state. However, in the Java port, more exclusive lines change to modified than shared, indicating that the exclusive state may be more useful in the Java version than in the Perl version.

We can see the overall impact of the exclusive state by looking at the origins of modified lines in the caches, shown in Figure 6. For each pair of bars in the above graphs, the first bar includes all sources of modified lines and the second bar includes only the write upgrades. First, we see that more than half of the lines become modified on write misses rather than upgrades. Interestingly enough, the number of modified lines that enter the cache on misses is highly dependent on the cache size and line size for Perl, but is mostly independent of the cache configuration for Java. A reason for this is that most of the write misses are due to the sharing in the Java version, which is largely from the shared memory lock. For relatively small caches, there are a large number of upgrades from exclusive to modified. In addition, for small caches, the Java implementation performs more upgrades from exclusive to modified than the Perl version. In particular, 62% of the upgrades are from the exclusive state in the Java Servlet, but only 24% of the upgrades are silent for the Perl CGI. However, for relatively large caches, there are few upgrades from the exclusive state, so it is less useful.

3.4. Dirty Lines

Figure 7 shows the MESI transitions for dirty cache lines. Note that the `M->M` is for modified lines that never leave the modified state. First, we see that a large number of modified lines downgrade to shared state due to another processor's read. As in the case of exclusive lines, we see that the behavior of the Perl version is highly dependent on the cache configuration, whereas the Java version is mostly independent of the configuration.

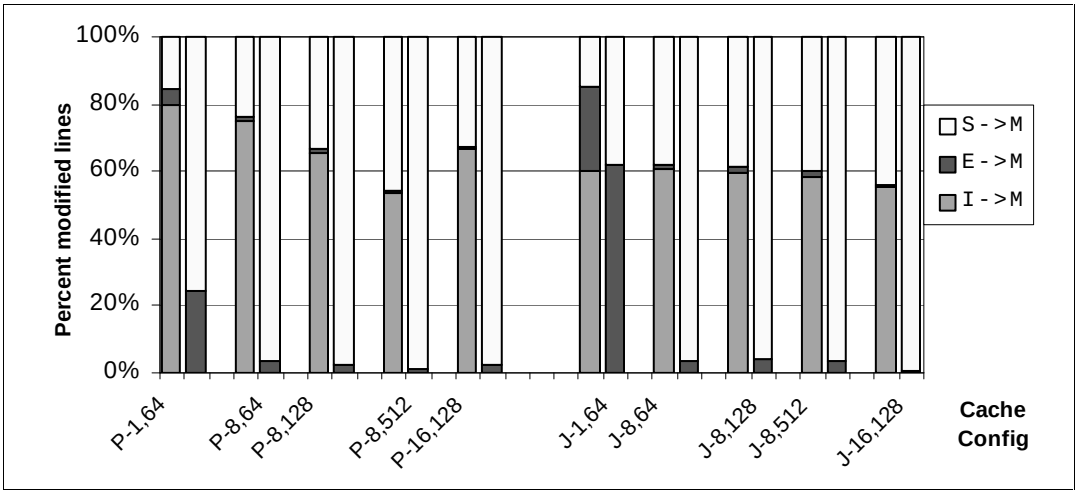


Figure 6: Sources of modified lines

Note that in each pair of bars, the first bar includes all sources of modified lines and the second bar only includes lines that upgraded from shared or exclusive state.

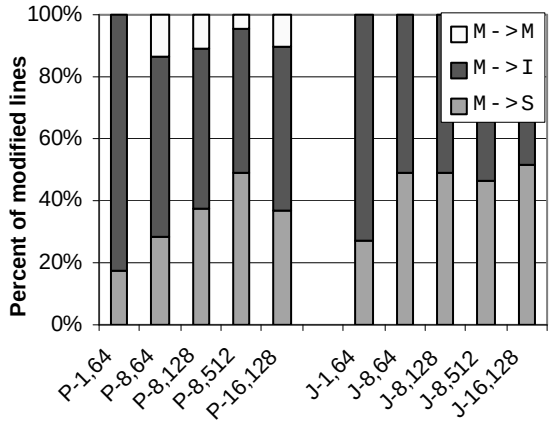


Figure 7: Transitions from modified lines

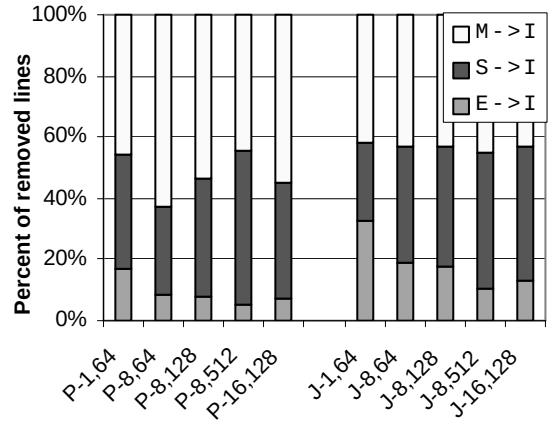


Figure 8: Lines removed from the cache

Similarly, we see that this is also true for the number of modified lines that are removed from the cache, as seen in Figure 8. Here, we see that about 40% to 65% of the lines that are removed from the cache are dirty, indicating that it is important to have an efficient mechanism for handling dirty removals. We also see that the number of shared lines increase with the cache size and line size, since it is more likely that a line will be shared as the caches become larger. Surprisingly, we see that the exclusive lines have the opposite behavior of the modified lines. In particular, the number of exclusive lines removed from the cache is relatively constant in the Perl version, but is dependent on the cache configuration in the Java implementation. This trend is also the opposite of the number of lines that enter the cache in exclusive state on read misses.

3.5. Stall Cycles

We show the number of stall cycles that each instruction has in Figure 9. Note that write back stalls occur when a processor has a second level cache miss after it has started performing a write back, so the processor cannot start handling the miss until the write back completes. In addition, on second level cache misses, the latency of the caches are included in the number of cycles of stall for the miss and not counted under second level cache hits. The restart processor stalls refer to a one processor cycle stall that occurs after the processor finishes handling a series of one or more first level cache misses, to model the overhead of restarting a processor after a cache miss. Lastly, the miscellaneous stall cycles contain the remaining stalls that the processors suffer, including first level data cache hazards and negative acknowledgements.

First, we see that the number of stall cycles is much larger for Java SPECweb99 than Perl, due to a larger cache miss rate. Note that the load of the Java

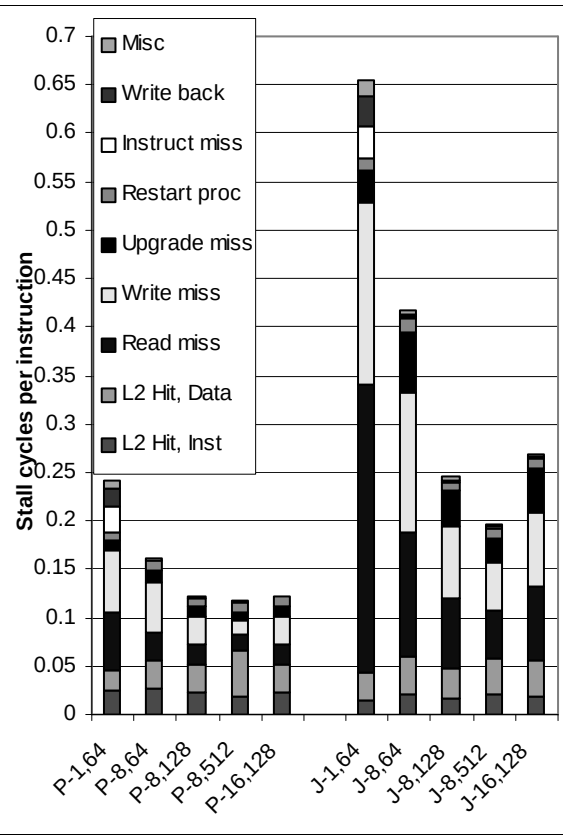


Figure 9: Stall cycles per instruction
Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

version is higher than the Perl version, so it is handling more transactions for the same amount of time. We also see that see that for both the Perl and Java versions, the number of stall cycles dramatically drops when one increases the cache from 1 MB to 8 MB and when one increases the line size from 64 bytes to 128 bytes. Our results show that the stall cycles due to instruction misses and write backs quickly disappear

when one increases the cache size to 8 MB. However, in the Perl version, the number of stall cycles does not change significantly when one increases the cache size to 16 MB or the line size to 512 bytes. In the Java implementation, though, the number of stalls decrease dramatically again when one increases the line size to 512 bytes. Surprisingly, in the Java version the stalls increase when one increases the cache size to 16 MB, due to an increase in the number of true sharing misses.

4. Processor Consistency

4.1. Implementation

To better understand the impact of consistency models on commercial workloads and whether consistency models warrant further study, we added simple implementations of sequential and processor consistency to our memory model. Under sequential consistency, all memory operations must appear to occur in the same order. For example, it is illegal for one processor to observe that a store occurs before a load and another processor to observe that the write occurs after the load[1]. To this end, our implementation of sequential consistency stalls each processor until all memory accesses complete for an instruction. By stalling the processor until all accesses complete, the processor cannot observe its own accesses before any other processor, since the accesses will have been broadcast over the bus.

Processor consistency relaxes memory ordering by allowing a processor to observe write to read ordering in a different order than another processor. In the previous example, processor consistency allows the first processor to observe the store before the load, even though the second processor observes the store after the load. Note that to ensure that programs work correctly, programmers may need to explicitly add serializations to their programs to ensure that a particular ordering of operations occur. In our simple implementation, under processor consistency, the processor stalls when a read miss occurs, but does not normally stall when a write miss occurs. Note that our simple implementation does not assign higher priority to reads than writes. Therefore, to avoid starvation, our implementation only allows there to be a maximum of 31 outstanding write misses, without stalling the processor. When the processor has more than 31 outstanding writes misses, it must stall until some of the misses complete and there are only 31 pending write misses.

[1]Note that it is legal for the load and store be performed in a different order with respect to different processors, as long as the two processors appear to observe that the operations occur in the same order.

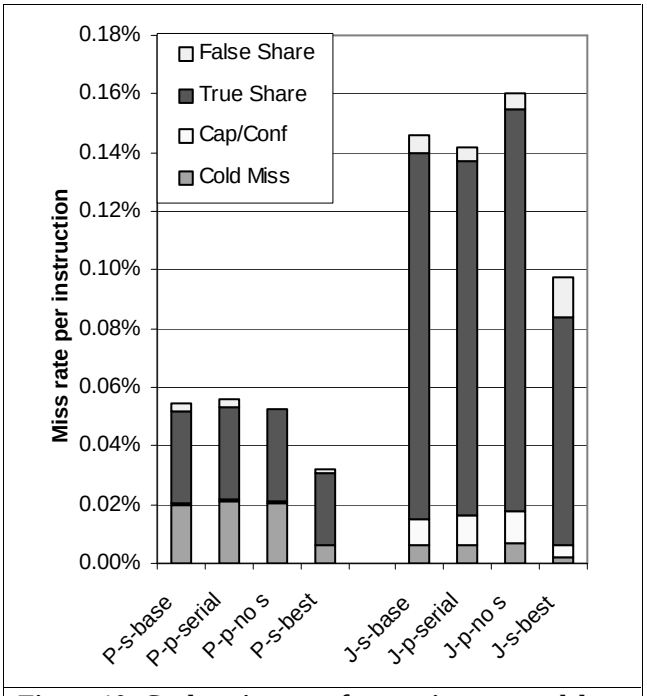


Figure 10: Cache miss rates for consistency models

X-s-base: baseline sequential consistency (8MB, 128 B)
X-p-serial: processor consistency with serializations
X-p-no s: processor consistency without serializations
X-s-best: sequential consistency with the highest performance cache configuration (8 MB, 512 B)

4.2. Results

In Figure 10, we show the second level cache miss rates for four different test cases for each workload. In the first test case, the simulated system uses sequential consistency with 8 MB second level caches and 128 byte cache lines (from Section 3). For the second case, the simulated system uses processor consistency with 8 MB second level cache with 128 bytes. The third test case shows the miss rate of a system that uses processor consistency, but is somehow able to ignore serializations and does not need to enforce the ordering that the programmer specified (this is similar to [15]). In the last test case, we show the miss rate of a system with sequential consistency, 8 MB second level caches, and 512 byte lines, which had the highest performance of all the sequential consistency cases (as seen in Section 3).

In the case of the Perl version of SPECweb99, we see that the miss rate of the system with processor consistency is slightly higher the system with sequential consistency. When a system with processor consistency does not need to perform serializations, the false sharing misses disappear.

Our results show that the interaction between the consistency model and miss rates is different for our Java implementation than Perl. In particular, we see that the number of sharing misses is smaller for processor consistency than sequential consistency. Surprisingly, when the system with processor consistency does not need to perform serializations, the number of sharing misses increase, such that the overall miss rate is 10% higher than in the case of sequential consistency. We believe that this occurs because there are some cases in which the serialization enforces that only one processor can use a cache line at a time. When the serialization is removed, then two or more processors use the cache lines at once, so they frequently steal the lines from each other before any of them can finish working with the line, increasing the amount of sharing.

Table 2 shows the number of serializations that occur in each workload. The processors do not need to add a memory barrier each time that they reach a serialization point. In particular, if there are no outstanding cache misses when a processor reaches a serialization point, then it does not need to insert a memory barrier. In addition, since our implementation of processor consistency does not prioritize reads over writes, if the last cache miss was for a read, then the processor does not need to add a memory barrier, since the processor will be stalled until all of the outstanding cache misses complete anyway. Note that due to current limitations in our simulator, we were not able to detect all serialization points for the Perl workload. Surprisingly, the processor does not need to add a memory barrier at most serialization points.

Figure 11 shows the number of stall cycles occur for each instruction for the four cases of memory consistency. The decrease in the number of stall cycles of the test cycles over the baseline (sequential consistency with 8 MB cache and 128 byte lines) is shown in Table 3. We see that adding processor consistency reduces the number of stall cycles for second level cache data hits, write misses, and upgrade misses by about 20% each, for the Perl version. Overall, the reductions in stalls create a respectable 13.68% overall improvement, which is much larger improvement than if one increases the cache line size to 512 bytes for sequential consistency. While removing the serialization points from processor consistency only improves the performance by another 2%, this shows that even a small number of serializations can have a significant impact on performance.

Our results that show that the Java version of SPECweb99 benefits from processor consistency more than the Perl version. In the Java version, the number of stalls for write misses and upgrade misses decrease by 47% and 43% respectively. However, the number of stall cycles for second level cache data hits decrease by only 2%. The reduction in stall cycles yields an overall 22.2% improvement, which is a little better than the improvement from increasing the cache line size to 512 bytes for the sequential consistency. Surprisingly, removing the serializations actually worsens the performance of the performance of the workload by 11%. The number of stall cycles for second level data misses, data read misses, write misses, and upgrade misses each increase by 10% to 16%. As

Workload	Memory Barriers Inserted	Serialization Points
Perl	22	?
Java	237	28,109

Table 2: Number of serializations

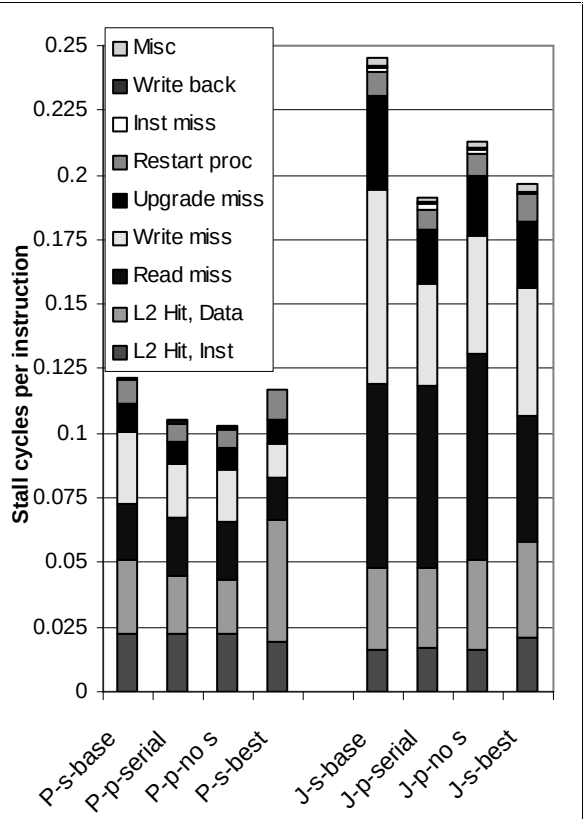


Figure 11: Stall cycles for consistency models

X-s-base: baseline sequential consistency (8MB, 128 B)
X-p-serial: processor consistency with serializations
X-p-no s: processor consistency without serializations
X-s-best: sequential consistency with the highest performance cache configuration (8 MB, 512 B)

	Perl Improve	Java Improve
Processor, serialization	13.68%	22.20%
Processor, no serialization	15.55%	13.43%
Sequential, best caches	3.38%	20.24%

Table 3: Decrease in stall cycles over baseline

mentioned before, the second level cache miss rate increases by 10% when the serializations are removed. Note however that even though the cache miss rate of processor consistency without serialization is higher than that of the baseline sequential consistency, it still manages to perform 13.43% better than sequential consistency.

5. Conclusion

In this work, using a functional execution driven full system simulator and a detailed memory model, we show that the memory system behavior of Java workloads is very different from that of Perl workloads. We find that our Java implementation of SPECweb99 suffers 169% to 213% more cache misses than the original Perl version.

Our results reveal several means for improving performance for both Perl and Java SPECweb99. First, for relatively large caches, increasing the line size is more effective for reducing the cache miss rates than increasing the size of the cache, even with very long line sizes. In addition, we find that when using relatively small caches (i.e., 1 MB second level caches), the exclusive state is effective at reducing stalls for write upgrades, by removing 24% of the upgrades for Perl and 61% of the upgrades for Java. However, the exclusive state is less useful for multiprocessor workloads with relatively large caches and is only able to remove about 1% to 5% of the upgrades. Note, however, that while the exclusive state is less useful for large caches in multiprocessor workloads, it is still important in the case of single processor workloads. Third, our simulations show that it is important to have an efficient cache to cache transfer mechanism for modified lines. For relatively large lines, 71% to 87% of the second level cache misses hit a modified line in another processor's cache for both the Java and Perl implementations of SPECweb99. Lastly, we find that relaxing the consistency model is effective at reducing the number of cycles that instructions are stalled and warrants further research. Our early results of consistency models show that the Perl CGI and the Java Servlet can improve performance by 13.68% and 22.2% by using processor consistency instead of sequential consistency. These results also show that if one attempts to optimize processor consistency by removing serialization points, one must be careful about which serialization points are removed. If one removes the wrong serializations, then the caches may suffer more cache misses, degrading overall performance.

Acknowledgments

We thank Shih-Chang Lai, Trung Diep, and the Virtutech support team for their valuable help with Simics. We wish to thank Yongjoon Lee for helping with the initial memory model. We also thank Harold Cain for the initial motivation for this work. This work was supported in part by NSF Grants CCR-0073440, CCR-0083126, and EIA-0103670.

References

[1] S. Adve, V. Pai, P. Ranganathan. "Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems." In *Proceedings of the IEEE: Special Issue on Distributed Shared Memory Systems*, pages 445-455. March 1999.

[2] H. Cain, R. Rajwar, M. Marden and M. Lipasti. "An Architectural Evaluation of Java TPC-W." In *Proceedings of The Seventh International Symposium on High-Performance Computer Architecture (HPCA-VII)*, January 2001.

[3] Q. Cao, P. Trancoso, J. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. "Detailed Characterization of a Quad Pentium Pro Server Running TPC-D." In *Proceedings of the Third International Symposium on High-Performance Computer Architecture (HPCA-III)*, February 1997.

[4] S. Dieckmann and U. Hölzle. "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks." In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999.

[5] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramaurthy, and P. Stenström. "The Detection and Elimination of Useless Misses in Multiprocessors." In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA 20)*, May 1993.

[6] K. Gharachorloo, A. Gupta, and J. Hennessy. "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 1991.

[7] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 1: Specification*, 1996.

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

[9] J. Kim and Y. Hsu. "Memory System Behavior of Java Programs: Methodology and Analysis." In *Proceedings of the 2000 International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS 2000)*, June 2000.

[10] K. Lepak, G. Bell, and M. Lipasti. "Silent Stores and Store Value Locality." In *IEEE Transactions on Computers*, Vol. 50, No. 11, November 2001.

[11] T. Li, L. John, N. Vijaykrishnan, A. Sivasubramaniam, J. Sabarinathan and A. Murthy. "Using Complete System Simulation to Characterize SPECjvm98 Benchmarks." In *Proceedings of ACM International Conference on Supercomputing (ICS 2000)*, May 2000.

[12] Y. Luo and L. John. "Workload Characterization of Multithreaded Java Servers." In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2001)*, April 2001.

[13] M. Martin, D. Sorin, H. Cain, M. Hill, and M. Lipasti. "Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing." In *34th Annual International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[14] V. Pai, O. Ranganathan, S. Adve and, T. Harton. "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[15] R. Rajwar and J. Goodman. "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution." In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[16] R. Radhakrishnan, N. Vijaykrishnan, L. John, and A. Sivasubramaniam. "Architectural Issues in Java Runtime Systems." In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture (HPCA-VI)*, January 2000.

[17] P. Ranganathan, V. Pai, and S. Adve. "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models." In *Proceedings of the Ninth Symposium on Parallel Algorithms and Architecture (SPAA-9)*, June 1997.

[18] B. Rychlik and J. Shen. "Characterization of Value Locality in Java Programs." In *Proceedings of the Workshop on Workload Characterization, ICCD*, September 2000.

[19] Y. Shuf, M. Serrano, M. Gupta, and J. Singh. "Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations." In *Proceedings of the 2001 International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS 2001)*, June 2001.

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

[21] Virtutech Corporation. Simics Full System Simulator. Information available at http://www.simics.com.

Characterizing TPC-H on a Clustered Database Engine from the OS Perspective

Yanyong Zhang[†], Jianyong Zhang[†], Anand Sivasubramaniam[†], Chun Liu[†], Hubertus Franke[‡]

[†] Department of Computer Science & Engineering
The Pennsylvania State University
University Park PA 16802
{yyzhang, jzhang, anand, chliu}@cse.psu.edu

[‡] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights NY 10598-0218
{frankeh}@us.ibm.com

Abstract

A range of database services are being offered on clusters of workstations today to meet the demanding needs of applications with voluminous datasets, high computational and I/O requirements and a large number of users. The underlying database engine runs on cost-effective off-the-shelf hardware and software components that may not really be tailored/tuned for these applications. At the same time, many of these databases have legacy codes that may not be easy to modulate based on the evolving capabilities and limitations of clusters. An indepth understanding of the interaction between these database engines and the underlying operating system (OS) can identify a set of characteristics that would be extremely valuable for future research on systems support for these environments. To our knowledge, there is no prior work that has embarked on such a characterization for a clustered database server.

Using a public domain version of a commercial clustered database server and TPC-H like[1] decision support queries, this paper studies numerous issues by evaluating performance on an off-the-shelf Pentium/Linux cluster connected by Myrinet. The execution profile clearly demonstrates the dominance of the I/O subsystem in the execution, and the importance of the communication subsystem for cluster scalability. In addition to quantifying their importance, this paper provides further details on how these subsystems are exercised by the database engine in terms of characteristics such as request sizes, spatial and temporal distributions. These characteristics provide insight on the benefits of possible optimizations in these subsystems. This includes the potential savings by avoiding copies across protection domains during I/O and the potential reduction in the number of messages by employing multicasts. Mechanisms for performing such optimizations are also discussed.

[1]These results have not been audited by the Transaction Processing Performance Council and should be denoted as "TPC-H like" workload.

1 Introductions

Clusters of workstations built with commodity processing engines, networks and operating systems are becoming the platform of choice for numerous high performance computing environments. Commodity hardware and software components make the price, upgradeability and accessibility of clusters very attractive, facilitating their widespread deployment in several application domains.

The speed at which clusters are being deployed for these diverse and challenging environments is outpacing the rate of progress in the systems software technologies and tools that are crucial building blocks for the applications. Most commodity off-the-shelf software (including the operating system) are not specifically tuned for cluster environments, and it is not clear if gluing together individual operating systems, that do not know the presence of each other, is the best approach to handle such loads. Further, off-the-shelf operating systems are meant to be for general purpose usage, with most of them really tuned for desktop applications or uniprocessor/SMP class server applications. Their suitability for cluster applications is not well understood. At the same time, one does not want to design/develop operating systems specifically for clusters, which would then go against the off-the-shelf rationale.

Just as many of today's operating systems (such as Linux) are not specifically customized for these emerging (some of these - like the database engines - are not really new, but are clustered implementations of the original version) applications on clusters, the applications are in turn not extensively tuned for these operating systems. An important reason is the fact that some of these, at least the database engines, have legacy codes that have evolved over several revisions/optimizations over the years, and it is difficult to fundamentally change their design overnight in light of these new systems (regardless of how modular they may be), which are still evolving. There is a substantial cost that is expended in testing/debugging to write specific software that exploits special features of the underlying system, and it is not clear if the ensuing rewards can offset this cost. The clustered versions of these legacy applications can be viewed more as an

exercise in porting, taking into account the new technologies and capabilities/limitations offered by the clusters. One cannot blame these developments since commercial vendors are often pressurized by several factors to get a product out of the door for a target platform as early as possible, and there is a need to support the product on several different platforms. Consequently, we have many legacy applications, such as the database engines which are the focus of this paper, running on operating systems that were not the initial targets of their implementation and on clusters which were not the initial target hardware. Over a period of time, revisions/improvements to the products are likely to address such concerns. There is visible evidence of this in the fact that there are ongoing research activities [3] from vendors, who already have commercial offerings, exploring alternate implementation styles

It is unavoidable to encounter such situations when technologies change, and it is unclear whether the application needs to be tailored/tuned for the underlying OS on a cluster, or vice-versa, or if we need to do a combination of the two. With the large code base of many of these legacy database engines, one could hypothesize that it would be easier to fine tune the operating system, especially with an open source OS such as Linux. A lot of work has already gone into optimizing the legacy applications, and the issues/optimizations may not be very different even for these new environments/OS.

This leads us to believe that there is the possibility of a middle ground, wherein if we know what issues are really important, then we could incorporate a few mechanisms/extensions in the OS and with a few user/configuration directives (or even slight application modifications) be able to enjoy the benefits of better matching the application with the underlying system. At the same time, such OS mechanisms/extensions may be rewarding for other applications as well and could very well become a feature of the OS in future offerings. Our goal in this paper is not to develop application-specific operating systems, nor is it to find out what OS mechanisms/capabilities are needed for extensibility/customization as other researchers have done [3]. Rather, coming from the applications viewpoint, we would like to make a list of recommendations based on the execution characteristics that can benefit future developments. We have also taken the liberty of suggesting possible mechanisms and their implementation (specifically in Linux) for optimizing the execution based on these gleaned characteristics. There are, arguably, other possible mechanisms/implementations for performing the same optimizations (even on Linux), or one could use these characteristics for customizing an extensible OS [2, 6] accordingly. Another possibility is to provide middleware that can better match the applications with the OS based on these characteristics.

In summary, a detailed characterization of the execution of applications on a cluster from the OS perspective can contribute to the knowledge-base of information that can be used for guiding future developments in systems software and applications for these environments. It would also be invaluable for fine tuning the execution for better performance and scalability, since each of these applications/environments has high commercial impact. In this paper we focus specifically on TPC-H queries, a decision-support database workload. This constitutes an important workload for business enterprises, with long running queries - ranging from a few minutes to a few days - that can benefit from the capabilities of a cluster.

It is well understood that I/O is the biggest challenge faced by database engines on uniprocessors/SMPs [7, 8, 9, 10] and there is a large body of prior work proposing hardware and software enhancements to address this problem. It is not clear if I/O becomes any less important when we move the engine to a cluster environment, since there is another factor to consider, which is the network communication. System scalability with cluster size is dependent on how parallel is the computation division across the cluster nodes, how balanced are the I/O activities on different nodes, and how does the communication traffic change with data set and cluster sizes. All this requires a careful profiling and analysis of the execution of the queries on the database engine. To our knowledge, that there has been no prior investigation of completely characterizing the execution of TPC-H on a clustered database engine, and studying these characteristics for optimization at the application-OS boundary.

Section 2 gives details on the experimental setup. Section 3 gives the overall system execution profile and the system scalability is examined in Section 4. Based on the system profile, the I/O and network characteristics and optimizations are discussed in sections 5, 6 and 7. Finally, Section 8 summarizes the results and contributions of this study.

2 Experimental Setup

TPC-H contains a sequence of 22 queries (Q1 to Q22), that are fired one after another to the database engine [2]. In this work, we consider response time for each query as main measure, i.e. the time interval between submitting the query and getting back the results.

All the tables are horizontally partitioned across the entire cluster using a hash-based scheme, and we have verified that this results in a balanced distribution across nodes. There is a client machine (not part of the cluster) that sends these queries to a database coordinator node on the cluster, which then distributes the work and gives back the results to the client. Each node of the cluster performs the queries on the rows residing on it and exchanges results via Myrinet if necessary. The client is connected to the cluster using Myrinet. As was mentioned, we run experiments on an 8 dual node Linux/Pentium cluster, that has 256 MB RAM and 18 GB disk on each node. The nodes are connected by both switched Myrinet and Ethernet, and we study these networks separately. We use Linux 2.4.8, which was the latest release at the time of conducting

[2] Q21 and Q22 take an inordinately long time and the results for these two queries are not included.

the experiments. (Please note that up to version 2.4.8. standard Linux kernels do not support raw disk IO interfaces.) This kernel has been instrumented in detail to glean different statistics, and also modified to provide insight on the database engine execution since we are treating it as a black box. We have also considered the overheads of instrumentation by comparing the results with those provided by the proc file system to ensure validity of what is presented here.Unless otherwise stated, the experiments use kernel level TCP over Myrinet for communication, and the dataset is 30 GB in size.

3 Operating System Profile

We first present a set of results that depict the overall system behavior at a glance. The following results have been obtained by both sampling the statistics exposed by the Linux proc file system (`stat`, `net/dev`, `process/stat`) as well as by instrumenting the kernel. The kernel instrumentation was done by inserting code in the Linux system call jump mechanism, as well as in the scheduler and points where there is pre-emption (such as blocking) or resumption. The proc file system information is used to present the percentage utilization of the system in different modes, the rates/frequency of I/O, page fault and network activities. The profile of different system calls is presented from the kernel instrumentation.

The results are shown in Table 1, which gives system statistics for each query in terms of: the percentage of time that the query spent executing on the CPUs in user mode (relative to its overall execution time), the percentage of time that the query spent executing on the CPUs in system mode (relative to its overall execution time), the average number of page faults incurred in its execution per jiffy (10 milliseconds in Linux), average number of file blocks read per jiffy, average number of file blocks written per jiffy, average number of packets sent over the network per jiffy, the average number of packets received from the network per jiffy, and the percentage utilization of the CPU(s) by the database engine during I/O operations (captures the overlap of work with I/O operations). The file block size is 4096 bytes, and the Maximum Transfer Unit (MTU) for network packets is 3752 bytes. In addition to these, the table also shows the top four system calls (in terms of time) exercised by each query during its execution, and the percentage of system time that is spent in each of these calls. These statistics help us understand what components of the OS are really being exercised, and the relative importance of these components.

From these results, we make the following observations:

- As is to be expected with database applications, the bulk of the execution time in the system mode is taken up by file system operations (pread/pwrite). These calls are employed to read and write the queried relational tables, as well as for any temporary tables that are needed along the way. Our examination of the execution leads us to believe that the considered database server goes via the file system for I/O accesses, and does not directly use raw disks or mmap operations.

  Disk operations are so dominating in some queries (Q1, Q8, Q12, Q17) that the CPU utilization does not cross 50% in these queries. I/O costs not only result in poor CPU utilization overall (because of waiting for disk operations to complete), but also in significantly increasing the system call overhead itself. Note that this system call overhead (system CPU time) does not include the disk latencies. Rather, this high overhead is due to memory copying, buffer space management and other bookkeeping activities. In some cases (such as Q12), the system CPU time (overheads) even exceeds the amount of time spent executing the useful work in the query at the user-level.

- Most of the I/O that is incurred is more due to reads than write operations. This is particularly characteristic of TPC-H queries, because most operations are for decision-support (requiring only reads).

- Though the numbers are not explicitly given here, we would like to point out that the high read overheads are not only because of the higher number of file system read calls, but are also due to the higher cost per invocation of this call. We noticed that a pread call can run to nearly a millisecond in some queries.Of all the system calls considered, we found the per pread invocation taking the maximum amount of time.

- When we examine the CPU utilization during I/O (last column of Table 1), we find that there is good overlap of work with disk activity in some queries. As we will point out later on in this paper, the bulk of I/O is initiated by the database prefetcher, which does not necessarily come into the critical path of the execution in many queries. However, queries such as Q12, encounter significant blocking.

- After the file system calls, we found socket calls (select, socketcall) to be the next dominant system overhead.

- Interprocess communication (IPC), though not as dominant as the other two OS components, does come in third in the overall system overheads.

- Despite the dominance of I/O in many queries, queries like Q11 have a high CPU fraction (particularly in the user mode). Even though there are I/O operations in these queries, their costs are overshadowed by useful work (CPU utilization is around 66% even during periods of disk activity, and the bulk of it is in the user mode). Another point to note from this observation, and in the fact that there is little variation in these results from node to node, is the hypothesis that such queries are likely to scale very well as we move to larger clusters since they can benefit from higher degrees of parallelism.

query	user CPU (%)	system CPU (%)	system CPU breakup (%)				page faults per jiffy	blocks read per jiffy	blocks written per jiffy	packets sent per jiffy	packets received per jiffy	CPU utilization during IO (%)
Q1	27.58	21.44	pread	pwrite	select	ipc	1.50	51.01	23.1151	0.0012	0.0015	26.87
			46.7	46.7	3.3	2.9						
Q2	56.22	15.67	socketcall	pread	select	pwrite	0.39	21.97	1.7718	1.9077	1.9248	53.73
			35.4	32.9	20.1	7.3						
Q3	40.76	17.76	pread	socketcall	select	pwrite	0.99	55.47	4.9591	0.9778	0.9938	55.40
			54.1	15.1	13.1	12.8						
Q4	51.48	15.19	pread	select	socketcall	pwrite	0.00	22.97	1.0478	0.3517	0.3652	68.41
			60.0	17.6	11.2	6.9						
Q5	58.96	15.68	socketcall	pread	select	pwrite	0.05	16.73	1.1369	2.0779	2.0849	42.81
			43.3	29.2	21.7	3.7						
Q6	40.65	22.20	pread	ipc	socketcall		1.73	90.72	0.0020	0.0012	0.0012	33.49
			90.1	4.9	4.3							
Q7	52.44	16.82	pread	pwrite	ipc	select	0.00	16.89	1.9467	2.3880	2.3521	31.04
			72.1	14.8	6.1	6.0						
Q8	20.65	17.71	pread	pwrite	select	ipc	0.01	27.63	4.8078	0.0261	0.0228	12.91
			59.5	23.7	9.4	5.8						
Q9	51.41	13.52	pwrite	pread	select	ipc	0.00	6.69	1.9276	0.0133	0.0136	23.21
			40.8	38.7	13.9	2.5						
Q10	41.79	17.87	pread	socketcall	select	pwrite	0.17	41.99	1.8880	0.8774	0.8859	18.71
			57.7	17.9	13.4	7.0						
Q11	81.00	13.28	socketcall	pread	select	ipc	0.49	18.87	0.0020	2.3794	2.4011	66.46
			43.6	27.0	24.8	3.9						
Q12	14.91	19.73	pread	selet	ipc		0.25	40.79	0.0197	0.0102	0.0101	4.8
			78.3	15.2	5.2							
Q13	53.23	21.86	pread	socketcall	select	ipc	1.62	45.86	0.0034	2.0966	2.0935	32.71
			45.7	30.3	18.8	4.6						
Q14	33.57	22.19	pread	select	ipc		1.01	84.78	0.0025	0.1156	0.1175	29.75
			85.6	8.3	5.1							
Q15	55.37	18.69	pread	select	ipc	nanosleep	0.60	75.26	0.0033	0.6260	0.7703	51.68
			71.7	14.0	10.9	1.9						
Q16	51.84	15.30	socketcall	pread	select	ipc	2.32	15.36	0.8454	2.4836	2.4993	46.24
			43.5	27.0	22.2	5.7						
Q17	23.71	18.26	pread	pwrite	select	ipc	0.00	32.76	5.2532	0.0036	0.0037	13.94
			59.4	26.1	8.7	4.9						
Q18	52.64	14.77	pread	socketcall	select	pwrite	0.04	17.16	0.6261	0.3890	0.3803	35.11
			61.1	17.3	13.3	5.2						
Q19	35.48	21.16	pread	select	ipc	socketcall	0.29	78.76	0.0032	0.3343	0.3329	23.38
			80.1	7.3	5.9	5.9						
Q20	56.98	14.72	pread	select	socketcall	ipc	0.00	15.74	0.1601	0.3456	0.2973	47.36
			53.4	25.6	15.3	3.6						

Table 1: System Profile (statistics are collected from node 1)

4 System and Workload Scalability

The previous experiments used the 8-node dual configuration on a 30 GB dataset, with Myrinet as the interconnect. We briefly present results to discuss the scalability of the execution as a function of number of nodes, the network (switched 1.26 Gbps Myrinet vs 100 Mbps Ethernet), and to study the trade-offs between a larger cluster vs a smaller one with more processors at each node (compare 4 node duals with 8 node uniprocessors), in Figure 1 for a 1 GB dataset. Results are normalized with respect to the 8 node dual configuration. It should be noted that a complete scalability study across the spectrum of parameters is well beyond the scope of this paper. Rather we are only trying to point out the relative performance and issues to understand the implications of some of these parameters.

We observe that increasing the number of nodes has several advantages. First, you get higher parallelism in computation and in I/O. The other significant benefit is in the ability to harness more physical memory (buffer space) across the nodes for the same dataset [16], which can reduce I/O further. The downside is the additional communication that may be incurred. We find that the benefits outweigh the drawbacks for nearly all queries except Q2 and Q17, even when we move from 1 to 2 nodes. This can be explained by the fact that Q2 has higher communication and Q17 has much lower CPU utilization that can benefit from parallelism (see Table 1). Moving on to 4 or 8 nodes, we find that in Q9, Q10, Q11, and Q20, overheads from parallelism hurt their performance as well. In nearly all other queries, we find significant savings (particularly at the smaller cluster sizes) from parallelism. A large portion of this savings is due to the higher buffer availability (reducing the total I/O, and not just providing parallel accesses to disks), that results in a superlinear performance improvement for queries such as Q9, Q11, Q12, Q15, and Q20. In general, we observe that 4 and 8 dual node configurations are good operating points for this dataset size of those considered. After a certain size, the savings from larger memory drop a little, with the overheads also offsetting a large portion of these savings and the gain in parallelism.

Next, when we compare the 4 node dual with the 8 uniprocessor nodes, we find that there is no clear winner. In Q1, Q3, Q4, Q5, Q6, Q8, Q12, Q14, Q17, Q18, and Q19, the uniprocessor node configuration is better, and the 4 node dual is better for the others. If you look at these queries in Table 1, these

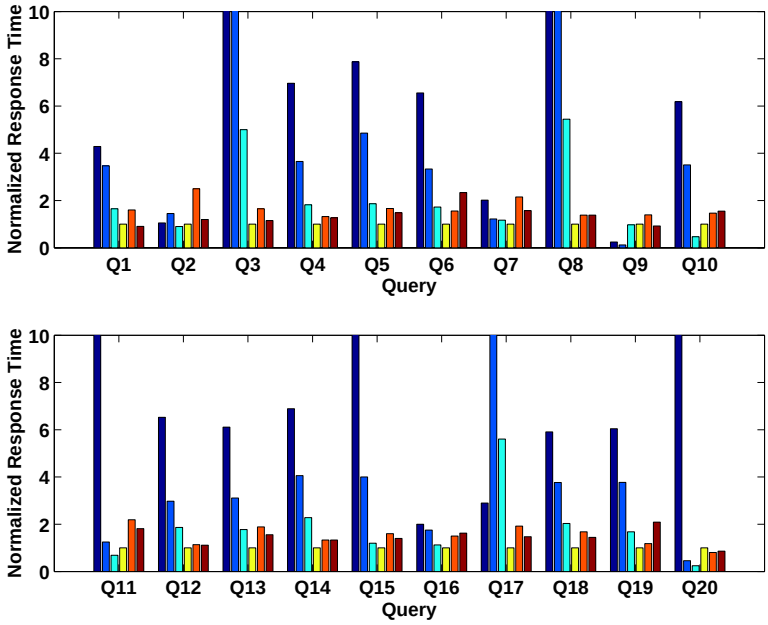


Figure 1: For each query, there are six bars (from left to right) showing the query response (execution) time under the following six configurations respectively: (i) one 2-way SMP node; (ii) two 2-way SMP nodes, myrinet; (iii) four 2-way SMP nodes, myrinet; (iv) eight 2-way SMP nodes, myrinet; (v) eight uniprocessor nodes, myrinet; and (vi) eight 2-way SMP node, 100Mbps Ethernet. We normalize the execution times under each configuration with respect to the time under configuration (iv). The dataset under examination is 1G. Note that the bars which reach the top of the graph actually exceed 10, and they are chopped in order to better show the differences between others.

have much lower CPU utilization, suggesting that disk activity is the bottleneck here. Even though both configurations provide the same number of CPUs, the 8 node configuration provides higher disk parallelism that benefits these queries. In the others, the communication is higher, and that becomes much more of a problem with a small 1 GB database considered in these experiments than the I/O parallelism.

It should be noted that in all these experiments we have used *constant problem size scaling* [4], with the physical memory increasing as we increase the cluster size. It would also be interesting to explore memory constrained scaling, or to consider a smaller cluster with more memory per node compared to a larger cluster with less memory per node (say keeping the overall cluster cost in dollars the same). Such issues are beyond the scope of this paper and we intend to explore these in the future.

When we move from a slow network (Ethernet) to a fast switched Myrinet platform, on the average query execution gets speeded up by 25%. We are not explicitly presenting results varying different dataset sizes, though we have conducted those experiments. In general, a larger dataset scales better as a function of the cluster size.

Before concluding this section, we would like to reiterate the importance of I/O and communication as we move to the future. Definitely, communication becomes important with larger clusters. At the same time, I/O can benefit significantly from such clusters not only because of the parallelism to disks, but also because of the higher memory capacities. Still, the I/O bottleneck would continue to pose challenges for clustered database services as dataset sizes increase. Further, I/O and communication become all that much more prominent with faster CPUs, and these are the focus of our attention in the rest of this paper.

5 I/O Subsystem: Characterization and Possible Optimizations

The results from the system profile clearly illustrate the importance of I/O for database servers. We now set out to look at the I/O subsystem more closely, trying to characterize its execution and look for possible optimizations.

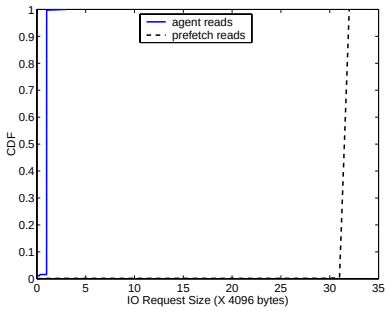
5.1 Characteristics

Table 2 sorts the queries in decreasing order based on the fraction of total query execution time spent in the pread system call obtained from earlier profile results. We can see that pread is a significant portion of the execution time in many queries. It takes over 10% of the execution time in 11 of the queries. It should be noted that this is the time spent in the system call (i.e. in buffer management, book-keeping, copying across protections domains, etc.), and does not include the disk costs itself. This implies that it is *not only important*

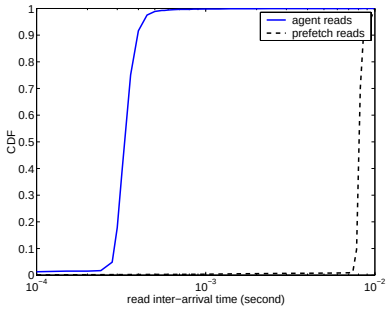
Query	Q6	Q14	Q19	Q12	Q15	Q7	Q17	Q8	Q10	Q1
% of exec. time	20.0	19.0	16.9	15.4	13.4	12.1	10.8	10.5	10.3	10.0
Query	Q13	Q3	Q4	Q18	Q20	Q2	Q9	Q5	Q16	Q11
% of exec. time	10.0	9.6	9.1	9.0	7.9	5.2	5.2	4.6	4.1	3.5

Table 2: pread as a percentage of total execution time

to lower or hide disk access costs, but to optimize the pread system call itself. In the interest of space, We focus on query Q6 which incurs the maximum pread overhead in the rest of this section. (the trends/arguments are similar for the others). Further, as with the profile results, we did not observe much variation across the nodes, and consequently examine the executions from the viewpoint of each node.



(a) CDF of read request size (Q6)



(b) CDF of inter-read time (Q6)

Figure 2: IO characterization results in terms of read request size and time between successive reads.

First, we would like to briefly explain how we believe the reads are invoked in the query execution. DB2 has several agent processes that actually perform the work in the queries, and prefetcher processes that make pread calls to bring in data from disk ahead of when the agent may need it. Figure 2 shows that the agent reads are more bursty, coming in closer proximity, than the reads issued by the prefetcher.

We found that the preads issued by the agent are usually for a block that has been recently read by the prefetcher just before that invocation. It may come as a surprise as to why this block could not have been serviced by the prefetcher directly (if it was only recently read), instead of going to the kernel. One possible explanation is that the agent is doing a write on this block, and it may not want to write into a page that is residing in the prefetcher. Instead of making instead of making ipc calls to remove the page from the prefetcher, it would be better to create a copy within the agent by using a pread call directly.

For further credibility on this hypothesis, before returning from pread calls, we modified the kernel to set the corresponding data pages to be read-only mode, and we found the agent to incur (write) segmentation faults (indicated as copy-on-write in Table 4) on nearly all those pages (compare the copy-to-user and copy-on-write columns for the agent in Table 4). Finally, it should be noted that the agent pread calls are much lower (both in terms of the number of calls and in terms of the number of blocks read) than those for the prefetcher.

We also include in Table 3 the fraction of pread block requests that hit in the Linux file cache for the prefetcher and the agents. As was pointed out, the agent requests come very soon after the prefetcher request for the same block, and thus nearly always hit in the Linux file cache. With the prefetcher requests on the other hand, we find the file cache hits range between 40-60%. We mentioned earlier that the prefetcher requests are usually for 32 blocks at a time. The Linux file cache manager itself does some read ahead optimizations based on application behavior and brings in 64 blocks (twice this size). With a lot of regularity (sequentiality) in I/O request behavior for this workload, this read ahead tends to cut down the number of disk accesses by around 50%.

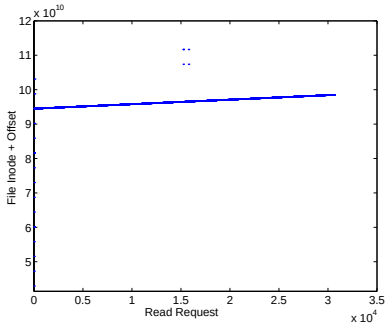
One can observe this regularity or patterns in the I/O request blocks by looking at Figure 3 (a) which shows the block number (expressed as a combination of inode + block number within file) requests that are issued to system. Further, Figure 3 (b) gives the same information for those requests that miss in the Linux file cache (on the average, every alternate request from (a) would miss here). One can visually observe regularity/sequentiality in both the requests that are generated and in the addresses that miss in the file cache. Prefetching and read-ahead are thus extremely useful for these executions.

5.2 Recommendations and Possible Optimizations

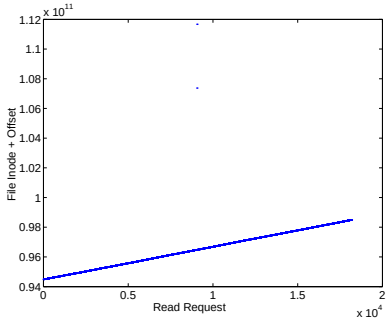
The previous subsection showed that system overheads in preads (not in the actual disk I/O) are a significant portion of the execution time. These read calls are usually for several blocks (32) by the prefetcher (that dominate the occasional single block reads by agent), and these blocks are hardly modified (TPC-H being a decision support workload, this observation is not very surprising). Further, many of these blocks are actually present already in the Linux file cache. We next explore how we can optimize the pread calls based on this

Query	prefetcher hit ratio	agent hit ratio	Query	prefetcher hit ratio	agent hit ratio
Q1	0.5711	1.0000	Q11	0.2788	1.0000
Q2	0.5321	1.0000	Q12	0.4735	1.0000
Q3	0.5164	1.0000	Q13	0.5261	1.0000
Q4	0.4873	1.0000	Q14	0.5344	1.0000
Q5	0.5991	1.0000	Q15	0.4227	1.0000
Q6	0.4729	1.0000	Q16	0.4409	1.0000
Q7	0.5182	1.0000	Q17	0.7365	1.0000
Q8	0.5300	1.0000	Q18	0.4226	1.0000
Q9	0.4683	1.0000	Q19	0.5625	1.0000
Q10	0.5082	1.0000	Q20	0.5453	1.0000

Table 3: Fraction of block requests that hit in Linux file cache for prefetcher and agent requests



(a) Request address (Q6)



(b) Page cache miss address (Q6)

Figure 3: pread request addresses and the corresponding addresses that miss in the Linux page cache (that are sent to disk)

information.

A significant portion of pread cost is expended in copying data (that is in a block in the file cache, either already or brought in upon disk I/O completion), from the kernel file cache to a user page, which needs to cross a protection-domain boundary (using the *copy-to-user()* mechanism). In the current 2.4.8 Linux implementation, a copy is actually made at this time.

This problem of reducing copying overheads for I/O has been looked at by several previous studies [14, 5, 12]. There are different techniques one could use, and a common one is to simply set the user page table pointer to the buffer in the file cache. This could affect the semantics of the pread operation in some cases, particularly when more than one user process reads the same block. In the normal semantic, once the copy is done, a process can make updates to it without another seeing it, while the updates would be visible without copies. This is usually addressed (as is by Linux in several other situations) by the *copy-on-write* mechanism, Some studies [5, 14, 12] suggest that even this may not be very efficient since updating virtual address mappings can become as expensive as copying. Instead, sharing of buffers between user and kernel domains is advocated. In this paper, we are not trying to advocate any particular technique for reducing these copies. Rather, we would like to find out what would be the benefit of reducing copy costs.

Remember, that a copy-to-user is actually needed when the pread is not page aligned and/or is interested in only part of the page. However, from our characterization results we find that most requests are page aligned and are in fact for an integral (32) number of papers. As a result, one could use the virtual remapping approach to implement the reduction of copies in these queries.

To examine the potential benefits of such an implementation, we track the total number of copy-to-user() calls that are made (actually one for each page) and the number of these calls that cannot be avoided (you cannot avoid it when there is a write segment violation and we need to do a copy-to-user at that time), during the execution of these queries after setting these pages to read-only mode. These numbers are shown in Table 4. As we can observe, the number of copy-on-writes that are actually needed is much lower than the number of copy-to-user invocations, as was suspected initially. In general, we get no less than 65% savings in the number of copies, with actual savings greater than 80% for most queries (see the last column of this table). Most of these savings are due to the prefetcher reads. Our measurements of copy-to-user routine for a single block using the high resolution timer takes around 30 microseconds for one page. For 32 block reads that the prefetcher issues, avoiding this cost can be a significant savings. This is particularly true when the blocks hit in the file cache (and there is no disk I/O) since this cost is a significant portion of the overall time required to return back to the application. Table 3 shows that this happens nearly 50% of the time. Even with disk activity, Table 1 shows CPU utilization

Query	prefetcher			agent			total
	copy-on-write	copy-to-user	% Reduction of copies	copy-on-write	copy-to-user	% Reduction of copies	% Reduction of copies
Q1	0	1040228	100	11551	11565	1.2	98.9
Q2	0	383334	100	63145	63145	0	85.7
Q3	0	1253107	100	52155	52157	0.003	96.0
Q4	0	997507	100	235758	235759	0.0004	80.9
Q5	0	1007919	100	307689	307689	0	100.0
Q6	0	914482	100	47	47	0	100.0
Q7	0	1084454	100	276790	276791	0.0003	79.7
Q8	0	978134	100	255057	255060	0.001	79.3
Q9	0	2478154	100	316500	316502	0.0006	88.7
Q10	0	974062	100	278213	278215	0.0007	77.8
Q11	0	170643	100	6833	6834	0.01	96.1
Q12	0	911544	100	134933	134933	0	87.1
Q13	0	184491	100	42	43	2.3	100.0
Q14	0	945619	100	38429	38430	0.003	96.1
Q15	0	1175166	100	38394	38395	0.003	96.8
Q16	0	36137	100	15001	15003	0.01	70.7
Q17	0	1968122	100	113962	113963	0.0008	94.5
Q18	0	1945777	100	502	503	0.19	100.0
Q19	0	865529	100	38429	38429	0	95.7
Q20	0	847755	100	50442	50444	0.003	94.4

Table 4: % of copy-to-user calls that can be avoided. Of the given copy-to-user calls, only the number shown under the copy-on-write are actually needed. The statistics are given for the prefetcher and agents separately, as well as the overall savings.

higher than 50% in most queries, suggesting that removing this burden of copying by the CPU would help query execution.

There is a caveat that we would like to point out with respect to the page remapping solution for reducing copying costs particularly with this database workload. With the prefetcher being quite active, and getting pages that are very often found in the Linux file cache, there is the possibility of very soon having a number of virtual address mappings to the file cache buffers (rather than to the buffers in the prefetcher itself). The file cache would then have to be made much larger (the buffer manager in the database engine uses several hundred megabytes of memory while the file cache is much smaller), or we will keep replacing entries in the file cache. File cache replacements may also need to be handled as copy-on-replacements, which can become a concern. These issues lead us to believe that a closer examination of the subtle interactions between the prefetching engine (that runs at user level) and the Linux file cache is needed, so that we understand the full ramifications of the pros and cons of these issues. Such a detailed exploration is well beyond the scope of this paper.

6 Network Subsystem: Characterization and Possible Optimizations

6.1 Characteristics

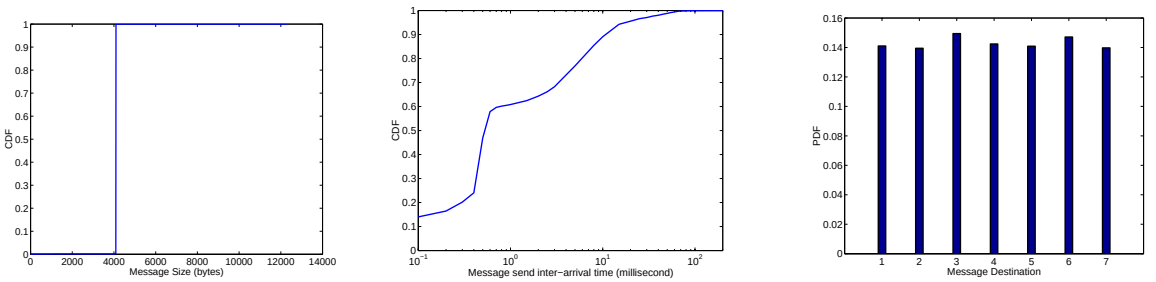
We next move on to the other exercised system service, namely TCP socket communication. As in the earlier section, we first attempt to characterize this service based on certain metrics that we feel are important for optimization. We examine the message exchanges based on the following characteristics: the message sizes, the inter-injection time (between successive messages by an application), and the destination for a message. We present these characteristics using density functions. The inter-injection time (or injection rate) and message size properties are captured by drawing their corresponding Cumulative Density Functions (CDF). The destination for a message is captured by a Probability Density Function (PDF) showing the probability of a message from a node heading to a specific node (7 possibilities on a 8 node cluster).

In the interest of space, We show the network subsystem characteristics pictorially in Figure 4 for query Q16, which has the highest message injection rates shown in Table 1. Many of the results are similar across queries, and we explicitly mention the differences in the text if there are any. These results have been obtained by instrumenting the kernel and logging all the socket events, their timestamps and arguments at the system call interface. As will be pointed out later on, for some characteristics we also needed to log messages themselves or at least their checksums.

From the density function graphs, we observe the following:

- The message length CDF graph shows that just a handful of message sizes are used by the database engine. In fact, we observed messages were usually either 56 bytes or 4000 bytes. We hypothesize that the shorter size (56 bytes) is used for control messages, and the larger size (4000 bytes) is used for actual data packets. Other message sizes were not very common. We found that Q1, Q6 and Q7 only send short messages, and short messages are the dominant part of Q15 communication (though there are a few long messages here as well). In the rest of the queries, we observed that there were around 40% short messages on the average.

- There are many messages that are sent out in close proximity (temporally). In fact, Figure 4 (b) shows that



(a)CDF of message size (Q16) (b) CDF of inter-arrival time (Q16) (c) PDF of message destination (Q16)

Figure 4: Characterizing Message Sends

nearly 60% of the messages are separated by less than 1 millisecond from each other temporally. In fact, the temporal separations are much lower for queries Q2, Q7, Q10, Q13, Q14, Q15, Q16, Q18 and Q19. We found similar observations for most of the other queries as well.

- The destination PDF graph is considerably influenced by the nature of database operations. In this engine, joins usually involve all-to-all communication of their corresponding portions of the table, and thus queries that are join intensive (such as Q11 and Q16 that are shown here) have the PDF evenly distributed across the nodes. There are a few queries, such as Q7, that are not really join intensive, but perform more specific operations that are based on values of certain primary keys. With such executions, there is a slight bias in communication towards nodes that have those values. Further, the engine uses a coordinator node that manages the query execution across the cluster, and this node performs a little different (we mentioned earlier that communication pattern was one issue that was different across nodes) than the others. In the other nodes, joins dominate most of the queries in general, and the communication load is more or less balanced.

6.2 Recommendations and Possible Optimizations

The above message characteristics say that messages are clustered together, often coming in close temporal proximity. Further, database operations such as joins use all-to-all communication of messages which have a high probability of being the same size. These observations suggest an hypothesis that many of these messages may actually be point-to-point implementations of a multicast/broadcast that the database engine would like to perform. It should be noted that a multicast can send the same information to several nodes at a much lower cost than sending individual point-to-point messages. This saves several overheads at a node (copies, packetization, protocol header compositions, buffer management, etc.) and can also reduce network traffic/congestion if the hardware supported it. Possible implementations of multicast are discussed later in this discussion.

To find out how many of the message exchanges can be modeled as multicasts, we investigated several approaches. During the execution, in addition to the above events, we also logged the messages themselves. These logs were then subsequently processed to compare whether successive messages were identical and addressed to different destinations. Another approach that we tried (which is actually a possible one to use within the OS during the course of execution itself to detect multicasts) is to compare checksums of successive messages. We would like to point out that we found that messages do indeed differ in the first 56 bytes, and that too in only 3 of these 56 bytes in most cases. After numerous experiments, we feel that the first 56 bytes are the header/control information, and the 3 bytes that really vary include the destination id and a possible sequence number. In our analysis of tracking the number of multicast possibilities, we ignore these three bytes, and check the differences for the rest to verify whether they contain the same information. If they do, then we identify that message as a multicast possibility (and the number of messages that would be incurred in a system that supports multicast would go down in this case). We found that both the approaches - actually comparing the messages or comparing the checksums - gave us similar results, and Table 5 gives the percentage of reduction in the number of messages that would be sent if the underlying infrastructure supported multicasts. This information is given for both the short and long messages to verify if multicasts are beneficial to any one class of messages or for both.

We find that there is a substantial multicast potential in these queries. There is a reduction in the total number of messages ranging from 8% to as high as 76%. In general, we find that the multicast potential is greater for the smaller (possibly control) messages than the longer (possibly data) messages. As was pointed out in the earlier results, both short and long messages are equally common in many queries, and we need to optimize both these classes.

This potential can be realized only if the underlying network supports multicasts (incidentally, we found a large number of these multicasts are in fact broadcasts, which Ethernet can support). Even assuming that the underlying infrastructure (either at the network interface level, or in the physical network implementation) supports multicast, the message ex-

query	% reduction of total messages	% reduction of small messages	% reduction of large messages	query	% reduction of total messages	% reduction of small messages	% reduction of large messages
Q1	44.7	71.4	38.7	Q11	9.6	28.6	0.1
Q2	20.4	58.7	0.2	Q12	8.3	7.8	2.9
Q3	48.2	64.3	38.0	Q13	24.5	75.2	0.1
Q4	22.6	58.6	0.1	Q14	27.9	80.4	0.7
Q5	8.0	7.1	8.4	Q15	46.6	56.5	0.7
Q6	76.4	78.6	45.5	Q16	59.1	63.0	56.9
Q7	57.5	71.4	56.2	Q17	41.5	66.7	27.3
Q8	29.1	75.5	4.8	Q18	11.4	32.3	0.00
Q9	66.8	78.5	61.1	Q19	26.7	79.4	0.2
Q10	25.0	73.6	0.1	Q20	21.1	62.8	0.1

Table 5: The potential impact of multicast on queries

changes should be injected into this infrastructure as multicast messages. This can be done at two levels. First, the application (i.e. the database engine) can itself inject multicast messages into the system. Our conversations with DB2 developers indicate that multicast is used by this database engine for purposes like replication, fault recovery etc., but not extensively for data exchanges when processing a query. Further, to work across numerous different platforms, sometimes it may be easier from the programming viewpoint for these applications to simply treat multicasts as point-to-point messages. The other approach, which we investigate, is to automatically detect multicast messages within the operating system (or middleware before going to sockets) and perform the optimizations accordingly. We next describe an online mechanism for such automatic detection. It should be noted that Table 5 gives an upper bound on message reductions by an offline analysis of the message traces, and the online version has only limited window of events to examine for detecting multicasts.

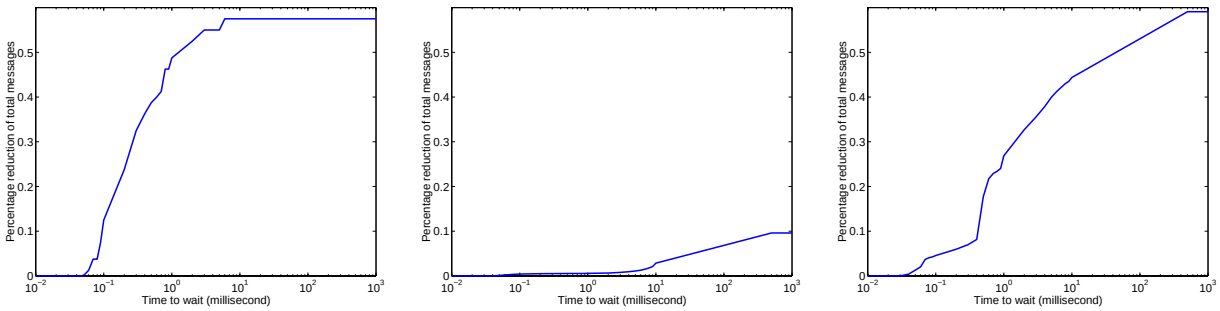
The online algorithm in the OS or middleware can make the system wait for a certain time window while collecting messages detected as multicasts, without actually sending these out. At the end of this window, we send a single multicast message for all the corresponding destinations of the saved messages. The advantage with this approach is that we do not send a message to a destination that the application does not send to. The drawback is that the time between successive messages and time window may be too long a wait that it may be better off just sending them as point-to-point messages. Further, if the window is not long enough, we may not detect some multicasts, and end up sending point-to-point messages.

Consequently, it is important to understand the impact of window size on multicast potential with this online algorithm. If we use such an algorithm within the OS/middleware, then the percentage reduction in the number of messages that need to be sent out with this approach is given in Figure 5 as a function of the time window that it waits for Q7, Q11, and Q16. As is to be expected, expanding the window captures a large fraction of multicasts until the benefits taper off. However, one cannot keep expanding a window arbitrarily since this can slow down the application's forward progress in case this message is needed immediately at the destination. We noticed that the TCP socket implementation on the underlying platform had one-way end-to-end latencies of around 100 microseconds. So it is not unreasonable to wait for comparable time windows since the message would anyway take a large fraction of that time to leave that node. If we consider, window wait times of say 500 microseconds, then we can see reduction of around 40% and 25% of the messages for Q7 and Q16 respectively. On the other hand, Q11 does not benefit much from such an online algorithm (nor from the offline algorithm). We found that queries Q6, Q7, Q9, Q15, Q16, Q17 (the graphs are not explicitly given here) had at least 15% message reduction with a wait time of 500 microseconds. It is, however, important to understand the ramifications of this wait time on query execution for eventual savings. Another point to note is that, if the send on one node and corresponding receive on another are not closely tied to each other (i.e. there is some temporal slackness), then one can increase the time window more aggressively. The underlying protocol can perhaps be extended to carry this slackness information back and forth to enable such decisions.

The observation about online monitoring of slackness, suggests that a more realistic implementation of this algorithm should make adaptive changes to the time window during the course of execution. As it finds that despite waiting, it is not able to combine messages as multicasts, it can adaptively decrease the window so that query execution does not get slowed down (in fact, it can do this even when it finds all the possible multicasts within a shorter time). Similarly, when the underlying protocol detects more slackness (this can be done at the receiver by examining the time difference between when the message gets in and when it is actually used), the algorithm can adaptively increase the window. The window may also need to be tuned based on the message size.

7 Optimizing I/O and Communication Simultaneously

One other issue that we considered for optimizing I/O and communication at the same time was the reduction of copies when there is the possibility of reading from disk and simply sending the data out to another cluster node. A system with a storage area network or a network attached storage disk (NASD) would facilitate direct access of remote data by a node, but the environment that we are conducting the eval-



(a) Q7 (b) Q11 (c) Q16

Figure 5: The impact of wait time on multicast message detection

uations on do not provide such capabilities. A node has to necessarily involve the remote CPU (specifically its counterpart database process) to retrieve data from its disk. Similarly the CPU has to be involved in the write to remote disk as well. Some of the optimizations that an OS could do, when there is no direct remote access hardware support, is to optimize the overheads that are involved when moving data from one I/O channel (say the disk) to another, is to manage buffers effectively by reducing copying costs [12, 5].

To evaluate the possibilities with this approach, we instrumented all the socket and I/O calls, and compared all the data that is read/written from disks and the socket messages. However, we did not find very close similarity in the content across these channels to suggest significant benefits from this approach. This can be attributed to the fact that several queries require a node to send out specific columns of a table after reading from disk. This usually requires retrieving all the columns of a row from the disk since it is stored in row-major order on disk, and then selecting the columns to send out. Sometimes, only select rows need to be sent based on some predicate. Such filtering/selection operations cannot be offloaded to Linux (to ask it to read from disk and perform these operations and then send the result out on sockets, thus reducing crossing protection boundaries and copying). Hence, we do not see too much scope for optimizing these mechanism unless the OS can allow extensibility/modularity to perform such operations. Earlier studies examining cross-channel buffer management [12] showed benefits for web servers, where the data is not really processed/filtered as is the case for these database engines. Further, in a cluster with hardware capabilities like a storage area network or NASD, it would be useful to incorporate intelligence at the disk so that these operations can be carried out to reduce transfer traffic as pointed out by others [11, 13, 15].

8 Summary of Results and Concluding Remarks

This is the first study to embark on a detailed characterization and to present a range of performance statistics for the execution of TPC-H queries on a medium sized Linux cluster of SMP nodes (a popular configuration in today's commercial market) connected by Myrinet and Ethernet. This has required distributing the tables of this workload across the disks of the cluster, implementing the queries, detailed kernel instrumentation to log events, and kernel modifications/extensions to better understand the interaction of the database server with the OS. A brief summary of the issues that are observed from the evaluation follows.

Moving from a uniprocessor/SMP to a cluster does not make I/O any less important for a database engine. We find that disk activity can push CPU utilization as low as 30% in some queries. The overhead of I/O is not just because of the disk latencies, and a significant portion is in the pread system call itself (copying costs mainly). Further, most of the I/O activity is because of the database prefetcher, that brings in large chunks of data (32 blocks at a time), ahead of use, and this is able to do a fairly good job because of reasonably good regularity/sequentiality in the queries. The read ahead feature of the Linux file cache, further helps in reducing the overheads providing hit rates of around 50%. While prefetching mechanisms, whether in the database engine or in the Linux file cache, can be tailored (it probably already is) for such sequentiality, the only consideration is the buffer space availability which in turn depends on physical memory availability. On the other hand, we find that it is extremely important to optimize the pread system call itself, by reducing the amount of copying. In this decision support workload that is read dominated, reducing copies can significantly reduce read overheads without sacrificing much on sharing costs. One could afford to pay higher penalties at writes if needed, if that can cut down read costs significantly. There are several known techniques for reducing copying costs , and in this study we examined the virtual address remapping scheme to show how many copies can actually be avoided. This scheme helps us achieve the objective without requiring any modifications to the legacy database code, but a more detailed investigation of the cost of address remappings is warranted since this can become expensive as pointed out earlier . It is our belief that asynchronous I/O provisioning in Linux [1] would also help, though this would again require application modifications.

The other system service that is also exercised is the socket communication to exchange control and data messages amongst the nodes. While this may not be as dominant as I/O, we find 5-10% of the execution time is spent in socket calls even for a 8-node cluster, and this issue will become more important for larger clusters. We find that many of these messages are identical, suggesting potential for multicasts/broadcasts, which the database engine implements as point-to-point messages.

It should be noted that our goal in this paper is not to recommend specific implementations or designs for improving performance. Rather, we are trying to identify characteristics of application-OS interactions and to suggest issues that can help improve performance for this workload.

Our ongoing work is examining how best to provide the support that the database engine requires, and how to manage the resources effectively in the presence of multiple users. In addition, we are investigating other TPC workloads, as well as other cluster applications such as web and multimedia services for similar studies.

References

[1] POSIX Asynchronous I/O. `http://oss.sgi.com/projects/kaio`.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th ACM. Symp. on Operating Systems Principles*, December 1995.

[3] J. Catozzi and S. Rabinovici. Operating System Extensions for the Teradata Parallel VLDB. In *Proceedings of Very Large Databases Conference*, pages 679–682, 2001.

[4] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[5] P. Druschel and L. L. Peterson. Fbufs: A highbandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, 1993.

[6] D. R. Engler, M. Frans Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM. Symp. on Operating Systems Principles*, December 1995.

[7] W. W. Hsu, A. J. Smith, and H. C. Young. Analysis of the Characteristics of Production Database Workloads and Comparison with the TPC Benchmarks. *IBM Systems Journal*, 40(3), 2001.

[8] W. W. Hsu, A. J. Smith, and H. C. Young. I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks - An Analysis at the Logical Level. *To appear in ACM Transactions on Database Systems*, 2001.

[9] M. A. Kandaswamy and R. L. Knighten. I/O Phase Characterization of TPC-H Query Operations. In *Proceedings of the 4th International Computer Performance and Dependability Symposium*, March 2000.

[10] K. Keeton. *Computer Architecture Support for Database Applications*. PhD thesis, Dept. of Computer Science, The University of California at Berkeley, Fall 1999.

[11] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, September 1998.

[12] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1), 2000.

[13] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active Disks for Large-Scale Data Processing. *IEEE computer*, 34(6):68–74, June 2001.

[14] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

[15] M. Uysal, A. Acharya, and J. Saltz. Evaluation of Active Disks for Decision Support Databases. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, 2000.

[16] S. Venkatarman. Global memory management for multi-server database systems. Technical Report CS-TR-1996-1325, Univ. of Wisconsin, 1996.