Box 35
folder 24

Performance analysis and engineering, 1986-1994

Digital Equipment Corporation records
Engineers' papers: Mike Uhler papers: Other systems and tech
X2675.2004, *Box 35; 36*

102749919

1 of 6

THE COMPUTER HISTORY MUSEUM
1 027 4991 9

Why Aren't Operating Systems Getting Faster As Fast As Hardware?

John Ousterhout

October, 1989

Abstract

This note evaluates several hardware platforms and operating systems using a set of benchmarks that test memory bandwidth and various operating system features such as kernel entry/exit and file systems. The overall conclusion is that operating system performance does not seem to be improving at the same rate as the base speed of the underlying hardware.

Copyright © 1989
Digital Equipment Corporation

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Table of Contents

1. Introduction 1
2. Hardware 1
3. Operating Systems 2
4. Kernel Entry-Exit 2
5. Context Switching 3
6. Select 4
7. Block Copy 4
8. Read from File Cache 5
9. Modified Andrew Benchmark 6
10. Open-Close 8
11. Create-Delete 8
12. Conclusions 10
13. References 10

List of Tables

Table 1: Hardware Platforms 1
Table 2: Getpid kernel call time 2
Table 3: Cswitch: echo one byte between processes using pipes. 3
Table 4: Time for select kernel call. 4
Table 5: Throughput of bcopy procedure for large blocks. 5
Table 6: Bandwidth of reading from the file cache 6
Table 7: The modified Andrew benchmark 7
Table 8: Modified Andrew benchmark, cont'd 7
Table 9: Time to open and close a file 8
Table 10: Time to create, use, and delete a file 9

1. Introduction

This technical note contains results from several benchmark programs that I ran recently on machines at DEC's Western Research Laboratory and at the University of California at Berkeley. The benchmarks are mostly ''micro-benchmarks'', meaning that each one measures a particular hardware or operating system feature (such as memory-to-memory copy speed or kernel entry-exit). Micro-benchmarks like these can identify particular strengths and weaknesses of systems, but they may not give a good indication of overall system performance. The benchmarks also include one ''macro-benchmark'', which exercises a variety of file system features and gives a better idea of overall operating system speed.

My goal in running the benchmarks was partly to compare different hardware platforms and operating systems, but mostly to understand whether or not operating system performance will scale with the base performance of hardware platforms. The answer for today's RISC workstations and RISC mainframes (including machines from DEC, Sun, and MIPS Computer) seems to be ''no'': when comparing slower older machines to newer RISC machines, the speedup in most of the benchmarks is much less than the difference in raw hardware speed.

The benchmarks suggest at least two possible factors that contribute to non-scalability of operating systems. The first is memory bandwidth, which has not scaled to match processor speed in faster machines. The second factor is file systems, some of which require synchronous disk I/Os in common situations. The synchronous I/O requirements limit the performance of operating systems when processors get faster but disks don't.

2. Hardware

I used five hardware platforms for the benchmarks, which are listed in Table 1. Table 1 also includes an abbreviation for each platform, which is used in the rest of this note, an approximate MIPS rating, and an indication of whether the machine is based on a RISC processor or a CISC processor. The MIPS ratings are my own estimates, and are intended to give a rough idea of the base integer performance provided by each platform. The main use of the MIPS ratings is to establish an expectation level for benchmarks. For example, if operating system performance scales with base system performance, then a DS3100 should run the various benchmarks about 1.5 times as fast as a Sun4 and about seven times as fast as a Sun3.

Hardware	Abbreviation	RISC/CISC	MIPS
MIPS M2000	M2000	RISC	20
DECstation 3100	DS3100	RISC	13
Sun-4/280	Sun4	RISC	9
VAX 8800	8800	CISC	6
Sun-3/75	Sun3	CISC	1.8
Microvax II	MVAX2	CISC	0.9

Table 1: Hardware Platforms

All of the machines were generously endowed with memory. As far as I know, no significant paging occurred in any of the benchmarks. In the file-related benchmarks, the relevant files all fit in the main-memory buffer caches maintained by the operating systems.

3. Operating Systems

I used four operating systems for the benchmarks: Ultrix, SunOS, RISC/os, and Sprite. Ultrix and SunOS are the DEC and Sun derivatives of Berkeley's 4.2 BSD UNIX, and are similar in many respects. RISC/os is MIPS Computer Systems' operating system for the M2000 machine. It appears to be a derivative of System V with some BSD features added. Sprite is an experimental operating system developed by my research group at U.C. Berkeley [3]; although it provides the same user interface as BSD UNIX, the kernel implementation is completely different. In particular, Sprite's file system is radically different from that of Ultrix and SunOS, both in the ways it handles the network and in the ways it handles disks. Some of the differences will be seen in the benchmark results.

The version of SunOS used for Sun4 measurements was 4.0, whereas version 3.5 was used for Sun3 measurements. SunOS 4.0 incorporates a major restructuring of the virtual memory system and file system; for example, it maps files into the virtual address space rather than keeping them in a separate buffer cache. This difference will also be reflected in some of the benchmark results.

4. Kernel Entry-Exit

The first benchmark measures the cost of entering and leaving the operating system kernel. It does this by repeatedly invoking the *getpid* kernel call. *Getpid* does nothing but return the caller's process identifier. Table 2 shows the average time for this call on different platforms and operating systems.

Configuration	Time (microseconds)	MIPS-Relative Speed
M2000 RISC/os 4.0	18	0.54
DS3100 Sprite	26	0.49
DS3100 Ultrix 3.1	25	0.60
8800 Ultrix 3.0	28	1.15
Sun4 SunOS 4.0	32	0.68
Sun4 Sprite	32	0.58
Sun3 Sprite	92	1.0
Sun3 SunOS 3.5	108	1.0
MVAX2 Ultrix 3.0	207	0.9

Table 2: Getpid kernel call time

The third column in the table is labeled ''MIPS-Relative Speed''. This column indicates how well the machine performed on the benchmark, relative to its MIPS-rating in Table 1 and to the Sun3 times in Table 2. Each entry in the third column was computed by taking the ratio of the Sun3 time to the particular machine's time, and dividing that by the ratio of the machine's MIP rating to the Sun3's MIP rating. For the UNIX-derivative operating systems (Ultrix, SunOS, and RISC/os) I used the Sun3 SunOS time; for Sprite I used the Sun3 Sprite time. For example, the MIPS-relative speed for the M2000 is (108/18)/(20/1.8) = 0.54. A MIPS-relative speed of 1.0 means that the given machine ran the benchmark at just the speed that would be expected based on the Sun3 times and the MIPS ratings from Table 1. A MIPS-relative speed less than one means that the machine ran this benchmark more slowly than would be expected from its MIPS rating, and a figure larger than 1 means the machine performed better than might be expected.

For the RISC machines, the MIPS-relative speeds in Table 2 are only about .5-.7. This indicates that the cost for entering and exiting the kernel has not improved as much in the RISC machines as their basic computation speed.

5. Context Switching

The second benchmark is called *cswitch*. It measures the cost of context switching, plus the time for processing small pipe reads and writes. The benchmark operates by forking a child process and then passing one byte back and forth between parent and child using pipes. Table 3 lists the time for each round-trip between the processes, which includes two context switches and one *read* and one *write* kernel call in each process. As with the getpid benchmark, MIPS-relative speeds were computed by scaling from the Sun3 times and the MIPS ratings in Table 1. Once again, the RISC machines didn't perform as well as might be expected, except for the DS3100/Ultrix combination.

Configuration	Time (ms)	MIPS-Relative Speed
M2000 RISC/os 4.0	0.30	0.71
DS3100 Ultrix 3.1	0.34	0.96
DS3100 Sprite	0.51	0.65
8800 Ultrix 3.0	0.70	1.0
Sun4 SunOS 4.0	1.02	0.47
Sun4 Sprite	1.17	0.41
Sun3 SunOS 3.5	2.36	1.0
Sun3 Sprite	2.41	1.0
MVAX2 Ultrix 3.0	3.66	1.3

Table 3: Cswitch: echo one byte between processes using pipes.

6. Select

The third benchmark exercises the *select* kernel call. It creates a number of pipes, places data in some of those pipes, and then repeatedly calls *select* to determine how many of the pipes are readable. A zero timeout is used in each *select* call so that the kernel call never waits. Table 4 shows how long each *select* call took, in microseconds, for three configurations. The first configuration used a single pipe that contained no data. The second configuration used 10 pipes, all empty, and the third configuration used 10 pipes all containing data. The last column is MIPS-relative speed again, computed using the "10 full" data. The performance of this benchmark was generally in line with the machines' MIPS ratings.

The M2000 numbers in Table 4 were surprisingly high for pipes that were empty, but quite low as long as at least one of the pipes contain data. I suspect that RISC/os's emulation of the *select* kernel call is faulty and is causing the process to wait for 10 ms even if the calling program requested immediate timeout.

Configuration	1 pipe (microseconds)	10 empty (microseconds)	10 full (microseconds)	MIPS-Relative Speed
M2000 RISC/os 4.0	10000	10000	108	0.84
DS3100 Sprite	76	240	226	1.04
DS3100 Ultrix 3.1	81	153	151	0.93
Sun4 SunOS 4.0	104	240	216	0.93
8800 Ultrix 3.0	120	265	310	.98
Sun4 Sprite	126	396	356	0.96
Sun3 Sprite	413	1840	1700	1.00
Sun3 SunOS 3.5	448	1190	1012	1.00
MVAX2 Ultrix 3.0	740	1610	1820	1.11

Table 4: Time for select kernel call.

7. Block Copy

The fourth benchmark uses the *bcopy* procedure to transfer large blocks of data from one area of memory to another. It doesn't exercise the operating system at all, but different operating systems differ for the same hardware because their libraries contain different bcopy procedures. The main differences, however, are due to the cache organizations and memory bandwidths of the different machines.

The results are given in Table 5. For each configuration I ran the benchmark with two different block sizes. In the first case, I used blocks large enough (and aligned properly) to use *bcopy* in the most efficient way possible, but small enough so that both the source and destination block would fit in the cache (if any). In the second case I increased the transfer size to be larger than the cache size, so that cache misses would occur continuously. In each case several transfers were made between the same source and destination, and the average bandwidth of copying is shown in Table 5.

Configuration	Cached (Mbytes/second)	Uncached (Mbytes/second)	Bytes/instruction
M2000 RISC/os 4.0	39	20	1.0
8800 Ultrix 3.0	22	16	2.7
Sun4 Sprite	11.1	5.0	0.55
DS3100 Sprite	10.2	5.4	0.43
DS3100 Ultrix 3.1	10.2	5.1	0.39
Sun4 SunOS 4.0	8.2	4.7	0.52
Sun3 Sprite	5.6	5.5	3.1
MVAX2 Ultrix 3.0	3.5	3.3	3.7

Table 5: Throughput of bcopy procedure for large blocks.

The last column in Table 5 is a relative figure showing how well each configuration can move large uncached blocks of memory relative to how fast it executes normal instructions. I computed this figure by taking the number from the second column (''Uncached'') and dividing it by the MIPS rating from Table 1. Thus, for the 8800 the value is (16/6) = 2.7. The most interesting thing to notice is that the CISC machines (8800, Sun3, and MVAX2) have normalized ratings of 2.5-4, whereas the RISC machines have ratings of 0.4-1.0. For the DEC and Sun RISC workstations, faster processors do not appear to have been accompanied by *any* increase in memory bandwidth. Thus, memory-intensive applications are not likely to scale on these machines. In fact, the relative performance of memory copying drops almost monotonically with faster processors, both for RISC and CISC machines.

8. Read from File Cache

This benchmark consists of a program that opens a large file and reads the file repeatedly in 16-kbyte blocks. For each configuration I chose a file size that would fit in the main-memory file cache. Thus the benchmark measures the cost of entering the kernel and copying data from the kernel's file cache back to a buffer in the benchmark's address space. The file was large enough that the data to be copied in each kernel call was not resident in any hardware cache. However, the same buffer was re-used to receive the data from each call; in machines with caches, the receiving buffer was likely to stay in the cache. Table 6 lists the overall bandwidth of data transfer, averaged across a large number of kernel calls.

The numbers in Table 6 reflect fairly closely the memory bandwidths from Table 5. The only noticeable difference is that the Sun4 does relatively better in this benchmark due to its write-back cache. Since the receiving buffer always stays in the cache, its contents get overwritten without ever being flushed to memory. For the DS3100, in contrast, the write-through cache causes information in the buffer to be flushed immediately to memory.

Configuration	Mbytes/second	MIPS-Relative Speed
M2000 RISC/os 4.0	15.6	0.45
8800 Ultrix 3.0	10.5	1.02
Sun4 SunOS 4.0	7.5	0.48
Sun4 Sprite	6.8	0.37
DS3100 Ultrix 3.1	4.8	0.21
DS3100 Sprite	4.4	0.16
Sun3 Sprite	3.7	1.0
Sun3 SunOS 3.5	3.1	1.0
MVAX2 Ultrix 3.0	2.3	1.48

Table 6: Bandwidth of reading from the file cache

9. Modified Andrew Benchmark

This is the one large-scale benchmark that I ran. It is a modified version of the Andrew benchmark developed by M. Satyanarayanan for measuring the performance of the Andrew file system [1]. The benchmark operates by copying a directory hierarchy containing the source code for a program, *stat*-ing every file in the new hierarchy, reading the contents of every copied file, and finally compiling the code in the copied hierarchy. In order to make the results comparable between different machines, I modified the benchmark so that it always uses the same compiler. In other words, regardless of which machine is executing the benchmark, the compiler is always the GNU C compiler generating code for a machine called SPUR.

The raw Andrew results are shown in Table 7. The table lists separate times for two different phases of the benchmark. The ''copy'' phase consists of everything except the compilation (all of the file copying and scanning), and the ''compile'' phase consists of just the compilation.

I ran the benchmark in both local and remote configurations. ''Local'' means that all the files accessed by the benchmark were stored on a disk attached to the machine running the benchmark. ''Remote'' means that as many files as possible were stored on a server machine and accessed over the network. In the Sprite and SunOS measurements, ''remote'' means that the benchmark was run on a diskless workstation, so absolutely all file accesses were remote. For the Ultrix measurements, temporary files used during compilation may have been stored locally even for the ''remote'' measurements. However, all of the files in the directory hierarchy being copied, and almost all of the program binaries, were stored remotely. For Ultrix and SunOS, NFS was used for remote accesses. For Sprite, Sprite's caching network file system was used (see [2] for details). In each of the remote cases, the server was the same kind of machine as the client.

Table 8 gives additional ''relative'' numbers: the MIPS-relative speed for local operation, and the percentage slow-down experienced when the benchmark ran with a remote disk instead of a local one. No ''Remote Penalty'' figures are given for the M2000 and 8800 because I wasn't able to benchmark them in a remote configuration.

Configuration	Copy (seconds)	Compile (seconds)	Total (seconds)
M2000 RISC/os 4.0 Local	13	59	72
DS3100 Sprite Local	22	98	120
DS3100 Sprite Remote	34	93	127
Sun4 Sprite Local	44	128	172
Sun4 SunOS 4.0 Local	46	130	176
Sun4 Sprite Remote	56	128	184
DS3100 Ultrix 3.1 Local	80	133	213
8800 Ultrix 3.0 Local	48	181	229
DS3100 Ultrix 3.1 Remote	115	154	269
Sun4 SunOS 4.0 Remote	108	162	270
Sun3 Sprite Local	52	375	427
Sun3 Sprite Remote	75	364	439
MVAX2 Ultrix 3.0 Local	214	1202	1416
MVAX2 Ultrix 3.0 Remote	298	1409	1707

Table 7: The modified Andrew benchmark

Configuration	MIPS-Relative Speed (Local)	Remote Penalty (%)
M2000 RISC/os 4.O Local	0.53	
8800 Ultrix	0.56	
Sun3 Sprite	1.0	3
DS3100 Sprite	0.49	6
Sun4 Sprite	0.50	7
MVAX2 Ultrix 3.0	0.60	21
DS3100 Ultrix 3.1	0.28	26
Sun4 SunOS 4.0	0.49	53

Table 8: Modified Andrew benchmark, cont'd

There are several interesting results in Tables 7 and 8. First of all, no operating system scaled to match hardware speedups. Second, Sprite comes out consistently faster than Ultrix or SunOS for remote access. Sprite shows hardly any performance degradation for remote access, and for the compilation phase Sprite was faster remote than local (I don't have a good explanation for why remote would be faster than local; at first I assumed that it was an experimental anomaly, but I have seen the effect on several different occasions). In contrast, NFS-based RISC workstations slow down by about 50% relative to local access. It appears to me that the relative penalty

for using NFS is increasing as machine speeds increase (for example, the MVAX2 slowed down by only 20% when using NFS instead of a local disk).

The third interesting result of this benchmark is that the DS3100-Ultrix combination is somewhat slower than would have been expected. For example, DS3100-Ultrix-Local is about 70% slower than DS3100-Sprite-Remote, and DS3100-Ultrix-Remote is not much faster than Sun4-SunOS-Remote.

10. Open-Close

I ran two other benchmarks in an attempt to explain the results in Table 7. The first of these is open-close, a benchmark which repeatedly opens and closes a particular file. The results are shown in Table 9 for two cases: a name with only a single element, and one with 4 elements. In the local case, the UNIX derivatives are consistently faster than Sprite. In the remote case Sprite is faster than SunOS, but slower than Ultrix. In any case, this benchmark cannot explain the differences in Table 7.

Configuration	"foo" (ms)	"/a/b/c/foo" (ms)
DS3100 Ultrix 3.1 Local	0.27	0.41
M2000 RISC/os 4.0 Local	0.32	0.83
Sun4 SunOS 4.0 Local	0.34	0.43
8800 Ultrix 3.0 Local	0.45	0.68
DS3100 Sprite Local	0.82	0.97
Sun4 Sprite Local	1.2	1.4
MVAX2 Ultrix 3.0 Local	2.9	4.7
DS3100 Ultrix 3.1 Remote	3.8	3.9
DS3100 Sprite Remote	4.3	4.4
Sun3 Sprite Local	4.3	5.2
Sun4 Sprite Remote	6.1	6.4
Sun3 Sprite Remote	12.8	16.3
MVAX2 Ultrix 3.0 Remote	36.0	36.9

Table 9: Time to open and close a file

11. Create-Delete

The last benchmark was perhaps the most interesting in terms of identifying differences between operating systems. It also helps to explain the results in Table 7. This benchmark simulates the creation, use, and deletion of a temporary file. It opens a file, writes some amount of data to the file, closes the file, then opens the file for reading, reads the data, closes the file, and finally deletes the file. I tried three different amounts of data: none, 10 kbytes, and 100 kbytes.

Table 10 gives the total time to create, use, and delete the file in each of several hardware/operating system configurations.

Configuration	No data (ms)	10 kbytes (ms)	100 kbytes (ms)
DS3100 Sprite Local	17	34	69
Sun4 Sprite Local	18	33	67
DS3100 Sprite Remote	33	34	68
Sun3 Sprite Local	33	47	130
M2000 RISC/os 4.0 Local	33	51	116
Sun4 Sprite Remote	34	50	71
8800 Ultrix 3.0 Local	49	100	294
Sun3 Sprite Remote	61	73	129
Sun4 SunOS 4.0 Local	66	830	940
DS3100 Ultrix 3.1 Local	80	146	548
MVAX2 Ultrix 3.0 Local	100	197	841
DS3100 Ultrix 3.1 Remote	116	370	3028
MVAX2 Ultrix 3.0 Remote	295	634	2500

Table 10: Time to create, use, and delete a file

This benchmark highlights a basic difference between Sprite and UNIX derivatives. In Sprite, short-lived files can be created, used, and deleted without any data ever being written to disk. Information only goes to disk after it has lived at least 30 seconds. In UNIX and its derivatives, the file system appears to be much more closely tied to the disk. Even with no data written in the file, the UNIX derivatives all required 35-100 ms to create and delete the file, regardless of the performance of the machine. This suggests that the creation and deletion operations are forcing data to disk and waiting for the disk operations to complete.

The create-delete benchmark also helps to explain the poor performance of DS3100 Ultrix on the Andrew benchmark. The basic time for creating an empty file is 60% greater in DS3100-Ultrix-Local than in 8800-Ultrix-Local, and the time for a 100-kbyte file in DS3100-Ultrix-Remote is 45 times as long as for DS3100-Sprite-Remote! The poor performance relative to the 8800 may perhaps be due to slower disks (RZ55's on the DS3100's); the poor remote performance is probably due to NFS's writing policy, which requires new data to be written through to disk when the file is closed. Note that DS3100-Ultrix-Remote achieves a write bandwidth of only about 30 kbytes/sec. This is almost twice as slow as I measured on the same hardware running an earlier version of Ultrix (3.0), and also about twice as slow as I have measured previously on Sun machines running SunOS and NFS.

Lastly, Table 10 exposes some suprising behavior in SunOS 4.0. Note that the time for a file with no data is 66 ms, but the time for 10 kbytes is 830 ms! This surpised me, so I also tried data sizes of 2-9 kbytes at 1-kbyte intervals. The SunOS time stayed in the 60-80ms range until the file size increased from 8 kbytes to 9 kbytes; at this point it jumped up to the 800-ms range.

12. Conclusions

In almost every benchmark the faster machines ran more slowly than I would have guessed from raw processor speed. In some cases, like getpid and cswitch, I don't have a good explanation for the discrepancy (the additional registers in the RISC machines cannot account for the difference all by themselves, for example). However, some of the benchmarks highlight issues for both hardware designers and operating systems people to think about.

On the hardware side, memory bandwidth has been allowed to slip relative to processor speed. If this trend continues, future machines (particularly workstations where cost considerations may tempt designers to skimp on memory system performance) are likely to be limited in performance by overall memory bandwidth. A fast cache may reduce the need for low-latency main memory, but it doesn't eliminate the need for high bandwidth in the main memory.

On the software side, operating system designers need to decouple file system performance from disk performance. Operating systems derived from UNIX use caches to speed up reads, but they require synchronous disk I/O for operations that modify files. If this coupling isn't eliminated, a large class of file-intensive programs will receive little or no benefit from faster hardware. Of course, delaying disk writes may result in information loss during crashes; the challenge for operating system designers is to maintain reliability while decoupling performance.

A final consideration is in the area of network protocols. In my (biased) opinion, the assumptions inherent in NFS (statelessness and write-through-on-close, in particular) represent a fundamental performance limitation. If users are to benefit from faster machines, either NFS must be scrapped (my first choice), or NFS must be changed to be less synchronous.

13. References

[1] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems* 6(1):51-81, February, 1988.

[2] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):134-154, February, 1988.

[3] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *IEEE Computer* 21(2):23-36, February, 1988.

WRL Research Reports

"Titan System Manual."
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

"Global Register Allocation at Link Time."
David W. Wall.
WRL Research Report 86/3, October 1986.

"Optimal Finned Heat Sinks."
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

"The Mahler Experience: Using an Intermediate Language as the Machine Description."
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

"The Packet Filter: An Efficient Mechanism for User-level Network Code."
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

"Fragmentation Considered Harmful."
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

"Cache Coherence in Distributed Systems."
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

"Register Windows vs. Register Allocation."
David W. Wall.
WRL Research Report 87/5, December 1987.

"Editing Graphical Objects Using Procedural Representations."
Paul J. Asente.
WRL Research Report 87/6, November 1987.

"The USENET Cookbook: an Experiment in Electronic Publication."
Brian K. Reid.
WRL Research Report 87/7, December 1987.

"MultiTitan: Four Architecture Papers."
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

"Fast Printed Circuit Board Routing."
Jeremy Dion.
WRL Research Report 88/1, March 1988.

"Compacting Garbage Collection with Ambiguous Roots."
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

"The Experimental Literature of The Internet: An Annotated Bibliography."
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

"Measured Capacity of an Ethernet: Myths and Reality."
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

"Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description."
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

"SCHEME->C A Portable Scheme-to-C Compiler."
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

"Optimal Group Distribution in Carry-Skip Adders."
Silvio Turrini.
WRL Research Report 89/2, February 1989.

"Precise Robotic Paste Dot Dispensing."
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

"Simple and Flexible Datagram Access Controls for Unix-based Gateways."
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

"Spritely NFS: Implementation and Performance of Cache-Consistency Protocols."
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

"Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines."
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

"A Unified Vector/Scalar Floating-Point Architecture."
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

"Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU."
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

"Integration and Packaging Plateaus of Processor Performance."
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

"A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance."
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

"Leaf: A Netlist to Layout Converter for ECL Gates."
Robert L. Alverson and Norman P. Jouppi.
WRL Research Report 89/12, July 1989.

"The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance."
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

"Long Address Traces from RISC Machines: Generation and Analysis."
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

"Link-Time Code Modification."
David W. Wall.
WRL Research Report 89/17, September 1989.

WRL Technical Notes

"TCP/IP PrintServer: Print Server Protocol."
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

"TCP/IP PrintServer: Server Architecture and Implementation."
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

"Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer."
Joel McCormack.
WRL Technical Note TN-9, September 1989.

"Why Aren't Operating Systems Getting Faster As Fast As Hardware?"
John Ousterhout.
WRL Technical Note TN-11, October 1989.

"Mostly-Copying Garbage Collection Picks Up Generations and C++."
Joel Bartlett.
WRL Technical Note TN-12, October 1989.

MARCH 1990

WRL Research Report 90/4

Virtual Memory vs. The File System

Michael N. Nelson

digital Western Research Laboratory 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word "`help`" in the Subject line; you will receive detailed instructions.

Virtual Memory vs. The File System

Michael N. Nelson

March, 1990

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

This paper examines the behavior of mechanisms for providing variable-size file data caches. It presents the results of running virtual-memory- and file-intensive benchmarks on the Sprite operating system [OCD88]; the benchmarks are designed to simulate real-life applications that represent the worst case for variable-size cache mechanisms. The results indicate that variable-size cache mechanisms work well when virtual-memory- and file-intensive programs are run in sequence; the cache is able to change in size in order to provide overall performance no worse than that provided by a small fixed-size cache. However, when interactive programs are run concurrently with file-intensive programs, variable-size cache mechanisms perform very poorly if file pages and virtual-memory pages are treated equally. In order to guarantee good interactive response, virtual memory pages must be given preference over file pages.

Table of Contents

1. Introduction 1
2. Providing Variable-Size Caches 2
3. Benchmarks 2
4. Why Not Just Used Fixed-Size Caches? 4
5. Variable-Size Cache Performance 5
    5.1. The ECD Benchmark 6
    5.2. The IFS Benchmark 8
6. Biasing Against the File System 9
    6.1 Implementation 9
    6.2 ECD Benchmark 10
    6.3 IFS Benchmark 10
    6.4. How Much of a Penalty 10
7. Conclusions 10
8. Acknowledgements 12
9. Bibliography 12

List of Figures

Figure 1: Elapsed Time and Utilization with Fixed-Size Caches 4
Figure 2: Mbytes Transferred with Fixed-Size Caches 5
Figure 3: Elapsed Time and Utilization Variable vs. Fixed 6
Figure 4: Mbytes Transferred Variable vs. Fixed 7
Figure 5: Elapsed Time and Server Utilization with Penalty 9
Figure 6: Network Traffic with Penalty 11

List of Tables

Table 1: Edit-Compile-Debug Benchmark 3
Table 2: Traffic between VM and FS 7
Table 3: IFS Benchmark 8
Table 4: IFS Benchmark with Penalty 12

1. Introduction

File data caches have been used in many operating systems to improve file system performance. In a distributed system the use of caches can reduce both network and disk traffic. A study of the use of caches on diskless workstations [NWO88] showed that the use of large caches can reduce the execution time of application programs by up to 1/3. Unfortunately, if file data caches are allowed to become too large, then they will conflict with the needs of the virtual memory system. In particular, if there is insufficient memory to run application programs, then the programs may slow down by factors of 10 to 100 because of excessive paging activity. Thus, if a cache is allowed to become too large, the improvement in file system performance may be more than offset by a degradation in virtual memory performance.

In order to provide both good file system performance and good virtual memory performance, several operating systems [BBM72, DaD68, Lea83, RaF86, Ras87] have implemented variable-size cache mechanisms. In these operating systems the portion of memory used for file data and virtual memory varies in response to the file and virtual-memory needs of the application programs being executed. These mechanisms will obviously work well when there is little or no contention for memory between file and virtual-memory pages.

This paper examines the behavior of variable-size cache mechanisms in the worst case; that is, the case when there is a serious amount of memory contention. It looks at both sequential and concurrent contention. Sequential contention corresponds to a user alternatively running programs that are either virtual memory or file intensive; an example of this type of contention is an edit-compile-debug loop where the editing and debugging are virtual-memory intensive and the compile is file intensive. Concurrent contention is when a user is running file intensive and virtual-memory intensive programs at the same time; an example of this type of sharing is a user interacting with a window system while a file intensive program is running.

I evaluated the behavior of variable-size cache mechanisms by running benchmarks that simulate both sequential and concurrent memory contention. The benchmark results indicate that variable-size cache mechanisms work well for sequential contention; the cache is able to change in size in order to provide overall performance no worse than that provided by a small fixed-size cache. However, when interactive programs are run concurrently with file intensive programs, variable-size cache mechanisms perform very poorly. The file intensive programs steal memory away from the interactive programs causing the interactive programs to exhibit poor response time because of extra page faults. In order to guarantee good interactive performance, virtual-memory pages should receive preferential treatment: a file page should not replace a virtual-memory page unless the virtual-memory page has been idle for a substantial period (e.g. 5 to 10 minutes).

The rest of the paper is organized as follows: Section 2 looks at mechanisms for providing variable-size caches including the mechanism provided in Sprite; Section 3 describes the benchmarks that were used to evaluate the variable-size caching schemes; Section 4 looks at the

performance of fixed-size cache schemes; Section 5 sees how well the variable-size cache schemes compare to fixed-size schemes; Section 6 examines the effect of giving virtual memory data preference over file data; and Section 7 offers some conclusions.

2. Providing Variable-Size Caches

The approach that has been commonly used to provide variable-size file data caches is to combine the virtual memory and file systems together; this is generally called the *mapped-file* approach. To access a file, it is first mapped into a process's virtual address space and then read and written just like virtual memory. This approach eliminates the file cache entirely; the standard page replacement mechanisms automatically balance physical memory usage between file and program information. Mapped files were first used in Multics [BCD72, DaD68] and TENEX [BBM72, Mur72]. More recently they have been implemented in Pilot [Red80], Accent [RaR81, RaF86], Apollo [LLH85, Lea83] and Mach [Ras87].

The Sprite approach to providing variable-size caches is quite different from the mapped-file approach. In Sprite, the file system and virtual memory system are separate. Users invoke system calls such as *read* and *write* to access file data. These system calls copy data between the file cache and the virtual address spaces of user processes. Variable-size caches are provided by having the virtual memory system and file system modules negotiate over physical memory usage. In this paper I will only give an overview of the Sprite mechanism; see [NWO88] or [Nel88] for more details.

In the Sprite mechanism, the file system module and the virtual memory module each manage a separate pool of physical memory pages. Virtual memory keeps its pages in approximate LRU order through a version of the clock algorithm [Nel86]. The file system keeps its cache blocks in perfect LRU order since all block accesses are made through the *read* and *write* system calls. Each module keeps a time-of-last-access for each page or block. Whenever either module needs additional memory (because of a page fault or a miss in the file cache), it compares the age of its oldest page with the age of the oldest page from the other module. If the other module has the oldest page, then it is forced to give up that page; otherwise the module recycles its own oldest page.

The advantage that the Sprite approach has over mapped-file approaches is that it makes it easy to discriminate between file and virtual-memory pages. This makes Sprite a good vehicle for running experiments in variable-size cache behavior. Of course, Sprite has the disadvantage that it requires more copies than mapped-file schemes. However, measurements presented in [Nel88] demonstrate that the extra copies have an insignificant impact on performance.

3. Benchmarks

In order to measure the performance of variable-size cache mechanisms, I developed two benchmarks and ran them on the Sprite operating system. The results obtained on Sprite should be similar to results obtained with the same benchmarks on systems with mapped-files.

The first benchmark that I used is an edit-compile-debug (ECD) benchmark that runs under the X11 window system on Sprite (see Table 1). This benchmark represents work that is commonly done on Sprite, and is both VM and FS intensive. Each program in the benchmark is run in sequence; no two programs are ever run concurrently. I used this benchmark to determine how well variable-size cache mechanisms and fixed-size cache mechanisms perform with

Phase	Description	FS I/O	VM Image Size
Edit	Run window-based editor on 2500 line file.	70 Kbytes	560 Kbytes
Compile	Compile VM Module	800 Kbytes	1 Mbyte
Link	Link the kernel	8 Mbytes	3 Mbytes
Debug	Run kernel debugger	4 Mbytes	8.5 Mbytes
Environment	The X window system plus several typescript windows and tools.	--	5 Mbytes

Table 1. The phases of the edit-compile-debug benchmark. The first two columns describe the phase of the benchmark. The third column gives the number of bytes read and written by each phase. The last column gives the size of the largest virtual memory image of the phase. The last row is not a phase in the benchmark but rather shows the total amount of memory required by the basic environment in which the benchmark is running.

sequential memory contention.

The ECD benchmark was run on a Sun-3/75 workstation with from 10 to 16 Mbytes of physical memory; if less than 10 Mbytes was used then the benchmark did not finish in a reasonable amount of time because of virtual-memory thrashing. The workstation's files were stored remotely on a Sun-3/180 file server with 16 Mbytes of memory†. Although the benchmark was executed on a diskless workstation, the results should be similar to results obtained by running the benchmark on a machine with a local disk. Each benchmark consisted of two runs through the edit-compile-link-debug loop. Each data point was taken from the average of three runs of the benchmark.

The other benchmark that I used runs a virtual-memory intensive program and a file-intensive program concurrently. This benchmark is used to determine the impact of variable-size cache mechanisms on interactive performance; I will refer to this benchmark as the IFS (Interactive-File-System) benchmark. The virtual-memory-intensive program is a program which periodically touches many pages in its virtual address space, dirtying some of them‡. This simulates a user who is interacting with a program. When a user interacts with a program, the program must have its code, heap and stack pages memory-resident in order to give good interactive response. In fact, if the user is interacting with a program under a window system such as X11, then several programs have to be memory-resident in order for the user to get good interactive response.

The file system component of the IFS benchmark is the UNIX sort program run on a 1-Mbyte file. The UNIX sort program is an external merge sort which uses many temporary files. The *sort* program is run concurrently with the interactive program to simulate a file system

† In this paper the term *client* will be used to refer to the diskless workstation that the benchmarks were run on and the term *server* will be used to refer to the workstation that stored the client's files.

‡ The fraction of memory that the interactive program dirties each time it touches the memory in its address space may impact the performance of the IFS benchmark. Measurements of 5 workstations running Sprite showed that between 40 and 60 percent of the memory that was being used by user processes was dirty. For this reason the virtual-memory intensive program dirties half of the pages that it touches.

program that attempts to grow its cache by stealing memory from an interactive program.

The IFS benchmark was run on a diskless 8-Mbyte Sun-3/75. The file server was a Sun-3/180 with 16 Mbytes of memory. 1.3 Mbytes of the 8 Mbytes were used by the kernel, which left 6.7 Mbytes for user processes. The interactive program used 5.7 Mbytes of memory and left at most 1 Mbyte for *sort* and the file system cache. This is small enough that *sort* will contend with the virtual memory system for memory.

4. Why Not Just Use Fixed-Size Caches?

The results from previous measurements of file data caches [NWO88] suggest that a large fixed-size cache will provide the best performance for file-intensive programs. However, for the two benchmarks used here, a small fixed-size cache is best. Figure 1 gives the elapsed time and server utilization for the ECD benchmark as a function of the amount of physical memory available on the client and the size of its file cache. A cache of 0.5 Mbytes provides the lowest elapsed time, and a cache from 0.5 Mbytes to 1 Mbyte gives the lowest server utilization for the benchmark; note that this benchmark is so virtual-memory intensive that even with the largest physical memory the smallest file cache is best.

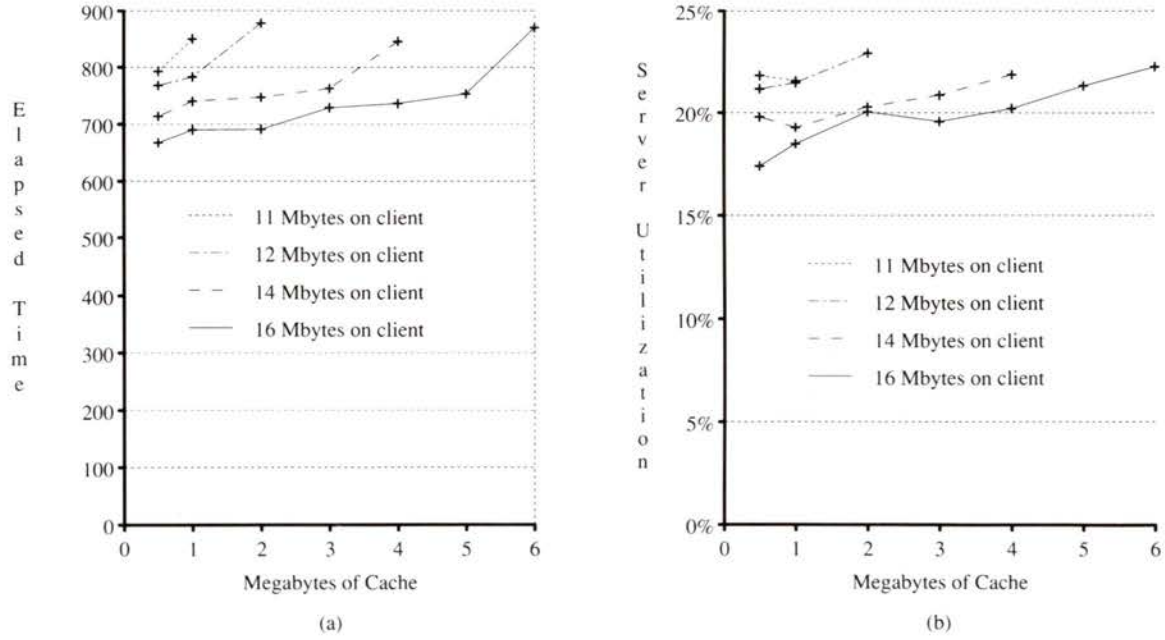


Figure 1. Elapsed time and server utilization for the edit-compile-debug benchmark with fixed-size caches as a function of client physical memory size. In both graphs the X-axis is the size of the client's file cache. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark. The system thrashed whenever the amount of physical memory left for the virtual memory system dropped below 10 Mbytes. I did not run the benchmark for points beyond where thrashing occurred (since elapsed time more than doubles), which explains why some curves have fewer data points than others.

Figure 2 clearly shows why the smallest cache is best for the ECD benchmark. As the cache grows in size, the number of file system bytes transferred drops. However, larger file caches leave less memory for virtual memory, so the number of page faults increases, resulting in more network traffic to fetch VM pages. This causes an increase in the total number of network bytes transferred and a corresponding increase in client degradation and server utilization.

A small fixed-size cache is also best for the IFS benchmark. This benchmark was designed so that interactive performance would degrade if more than 1 Mbyte were used for the file system cache. In addition, measurements in [NWO88] show that even with a small cache the *sort* benchmark will only execute at most 25 percent more slowly than with a large cache. Thus a small fixed-size cache will give instantaneous interactive response (no page faults required) while only slightly degrading file system performance.

The results with fixed-size caches demonstrate that different cache sizes are needed for different types of programs. The results in [NWO88] show that when purely file-intensive programs are run, a large cache is best. However, when a mix of file- and virtual-memory-intensive programs are run, then a small cache is best.

5. Variable-Size Cache Performance

It is clear from the previous section that different file cache sizes are required for different types of programs. Variable-size cache mechanisms attempt to provide the ability to adjust the

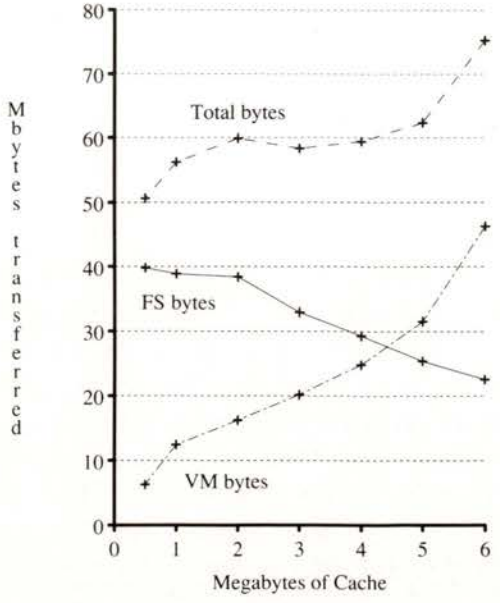


Figure 2. This graph gives the number of Mbytes transferred across the network for the edit-compile-debug benchmark with fixed-size caches and 16 Mbytes of memory on the client. Graphs of network bytes transferred for the other four memory sizes yield similar results. The X-axis is the size of the cache and the Y-axis is the number of Mbytes transferred. The "FS bytes" line is the amount of file system data transferred across the network, the "VM bytes" line is the amount of virtual memory data transferred across the network, and the "Total bytes" line is the total amount of network bytes transferred which includes file and virtual memory data as well as packet headers and control packets.

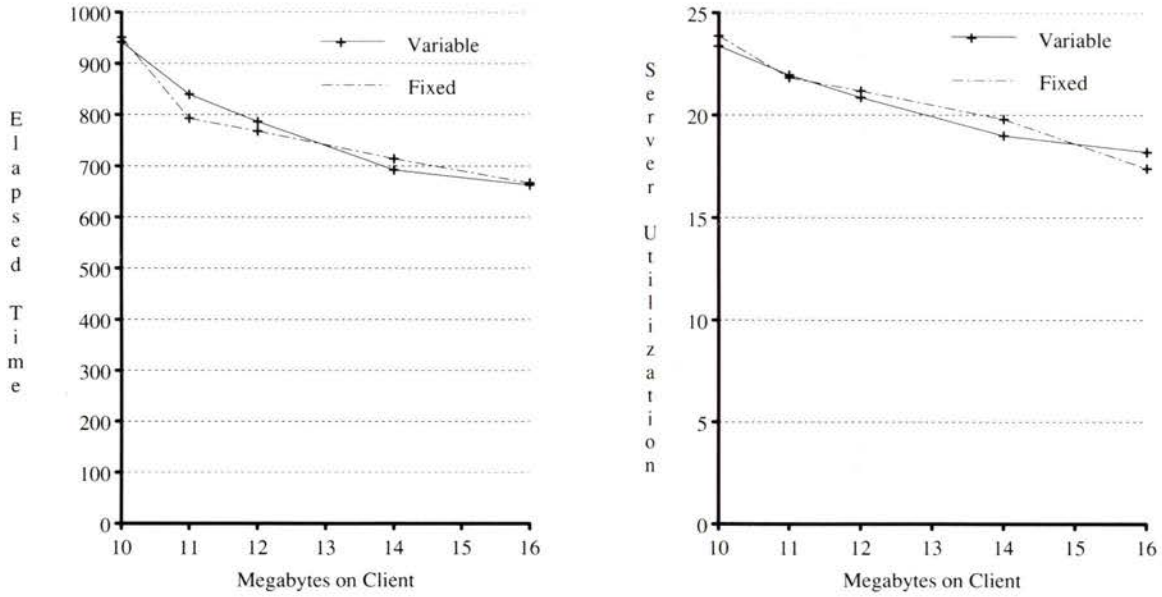


Figure 3. Elapsed time and server utilization for the edit-compile-debug benchmark with a variable-sized cache and with the smallest fixed-size cache as a function of physical memory size. In both graphs the X-axis is the amount of cache. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark.

size of the cache based on the types of programs that are being run. This section examines how the Sprite variable-size cache mechanism affects the performance of the ECD and IFS benchmarks. Measurements of other variable-size cache mechanisms should yield similar results.

5.1. The ECD Benchmark

For the ECD benchmark, variable-size cache mechanisms work quite well. Figure 3 shows that, in terms of elapsed time and server utilization, the variable-size and fixed-size cache mechanisms provide nearly identical performance. The reason why the performance is similar is demonstrated in Figure 4, which gives the amount of network traffic. The variable-size cache gives consistently fewer file system bytes transferred than a fixed-size cache, and the fixed-size cache gives fewer virtual memory bytes transferred. However, in terms of total bytes transferred, the variable-size cache is slightly better than the best fixed-size cache. Thus, the poorer virtual memory performance for the variable-size cache is more than offset by the much better file system performance.

The edit-compile-debug benchmark shifts between file- and virtual-memory-intensive programs. This requires that there be constant shifts in the allocation of physical memory between the file system and the virtual memory system (see Table 2). The minimum and maximum cache size columns from Table 2 show that the file cache varied widely in size during the life of the benchmark, going from the minimum possible size (0.25 Mbytes) up to over half the amount of physical memory available. As the amount of physical memory increased, the maximum size of the cache increased as well; the variable-size cache mechanism allowed the file system to take

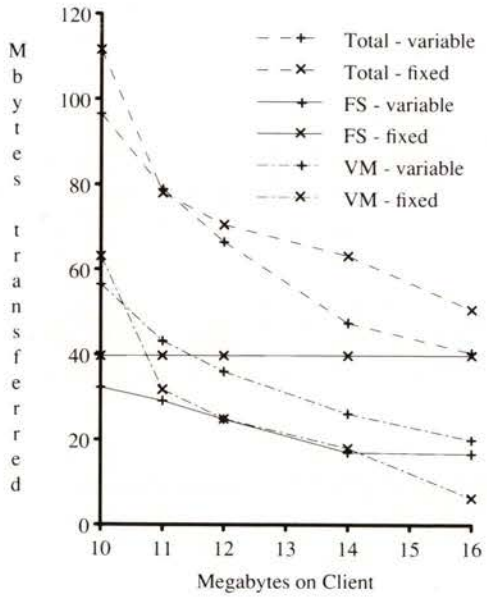


Figure 4. This graph gives the number of Mbytes transferred across the network with variable-size and smallest-fixed-size caches and 16 Mbytes of memory on the client. The X-axis is the amount of cache and the Y-axis is the number of Mbytes transferred. There are two lines for file, virtual memory and total bytes transferred: one for variable-size caches and one for fixed-size caches. The total bytes transferred lines includes file and virtual memory data bytes plus bytes from packet headers and control packets.

Client Mem (Mbytes)	Min Cache Size (Mbytes)	Max Cache Size (Mbytes)	FS Asks VM		VM Asks FS	
			Num	Satisfied	Num	Satisfied
10	0.25	5.6	8125	1810	2942	1846
11	0.25	6.4	7105	1889	2610	1967
12	0.25	6.9	5840	1964	2555	2075
14	0.25	8.7	4012	1957	2669	2162
16	0.34	8.8	3652	1937	2629	2229

Table 2. Traffic between the virtual memory system and the file system. The first column gives the amount of physical memory available on the client. The second and third columns give the minimum and maximum file cache sizes during the benchmark. The fourth and fifth columns are the number of times that the file system asked the virtual memory system for the access time of its oldest page and the number of times that it was able to get a page from the virtual memory system. The sixth and seventh columns are the same as the previous two, except that they are the number of times the virtual memory system asked the file system for memory.

advantage of the extra physical memory.

Table 2 also quantifies the negotiation between the virtual memory system and the file system. As the amount of physical memory increased, the number of times that the file system attempted to get memory from the virtual memory system dropped dramatically; however, the number of times that the file system was successful in stealing a page from the virtual memory system remained fairly constant across all memory sizes. In contrast, the number of requests for

memory made by the virtual memory system to the file system remained reasonably constant for all memory sizes, but the virtual memory system was more successful in taking pages from the file system as the amount of memory increased.

Table 2 suggests that the virtual memory system is much less elastic in its needs than the file system, at least for this benchmark; I hypothesize that this is true in general. The low success rate that the file system has when asking the virtual memory system for memory implies that the pages in the virtual memory system are being more actively used than those in the file system. Thus, the virtual memory system has fairly strict memory needs regardless of the physical memory size, and it actively uses the pages that it has. On the other hand, the file system caches files after they are no longer being used so it will grow to fill the available memory. Since the file system does not actively use many of its cached pages, its pages are the best candidates for recycling.

5.2. The IFS Benchmark

Variable-size caches work poorly for the IFS benchmark (see Table 3). The performance is dependent on the length of the interactive program's sleep interval. Short sleep intervals correspond, for example, to temporary pauses in an editing session. Long sleep intervals correspond, for example, to windows that have been idle because the user was working in a different window.

Table 3 shows that the interactive response time has a high variance: sometimes it is instantaneous and other times it takes up to 22 seconds. This corresponds to a user typing a key stroke and waiting 22 seconds for a response from the program. The response time gets worse as the sleep interval is increased. Longer sleep intervals allow the sort program to steal more memory which causes the interactive program to wait for pages to get faulted in from the file server.

In addition to producing poor interactive response, the use of variable-size caches also degrades the performance of the *sort* benchmark. The benchmark takes up to 72% longer to execute than the standalone case, and 50% longer than when a small fixed-size cache is used. The performance degrades because the CPU is busy trying to fault in pages for the interactive

Sleep	Response Time			Sort Time		Page	Page	Cache Size	
Interval	Min	Max	Avg	Time	Deg	Ins	Outs	Min	Max
1	0.0	4.7	0.1	79.6	33%	173	456	152	784
5	0.0	4.5	0.8	83.8	40%	437	509	157	842
10	1.9	13.3	5.9	103.4	72%	1605	1177	146	1226
30	12.5	22.0	15.8	96.3	61%	1250	983	141	2533

Table 3. Results for the IFS benchmark with variable-size caches. Each data point is the average of the results from three runs of the benchmark. The first column gives the number of seconds that the interactive benchmark slept before touching all of its memory. Columns 2 through 4 give the minimum, maximum and average number of seconds it took the interactive benchmark to touch all of its memory when it awoke from its sleep. Columns 5 and 6 give the total number of seconds it took to execute the *sort* benchmark, and the amount of degradation relative to the standalone case which took 60 seconds. Column 7 is the number of pages read in from swap files and Column 8 is the number of pages written to swap files. Columns 9 and 10 give the minimum and maximum amount of memory in the cache in Kbytes.

benchmark; if the interactive benchmark is memory resident, then it utilizes very little of the CPU.

6. Biasing Against the File System

The results of the IFS and ECD benchmarks demonstrate that virtual memory performance is the most important factor in determining overall system performance. This was shown with the ECD benchmark where performance degraded as the size of fixed-size file cache was increased; fortunately the variable-size cache mechanism worked well in this case and was able to adjust the size of the cache effectively. Unfortunately, the variable-size cache mechanism did not work well for the IFS benchmark. These results indicate that it may not be practical to treat virtual memory and file data equally in a variable-size cache mechanism. This section evaluates the effect of giving virtual-memory pages preference over file pages.

6.1. Implementation

Since the Sprite mechanism treats virtual memory and file data separately it is quite easy to implement a scheme that biases against the file system. The method used in Sprite involves adding a fixed number of seconds to the reference time of each virtual memory page. This makes each virtual memory page appear to have been referenced more recently than it actually was. For example, if 5 minutes is added to the reference time of each virtual memory page, then the file system will not be able to take any page from the virtual memory system that has been

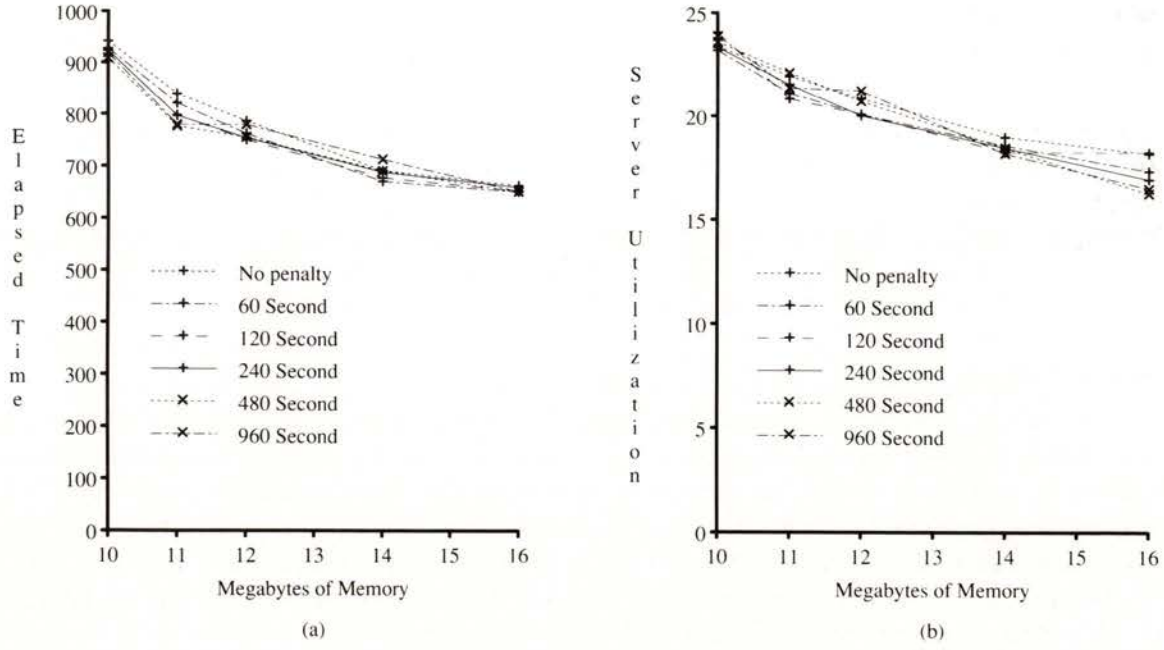


Figure 5. Elapsed time and server utilization with various penalties as a function of client physical memory size. In both graphs the X-axis is client memory size. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark.

referenced within 5 minutes of the oldest file system page.

6.2. ECD Benchmark

Penalizing the file system has little or no effect on the performance of the ECD benchmark. Figure 5 shows that, regardless of the penalty, the elapsed time and server utilization are about the same. Figure 6 shows why the penalty has no effect. As the penalty is made larger, the virtual memory performance gets better and the file system performance worse. The result is that overall performance is about the same regardless of the penalty.

6.3. IFS Benchmark

Penalizing the file system is very effective in improving the performance of the IFS benchmark. Table 4 shows that the interactive response is excellent when the file system is penalized. The 120-second penalty prevents the file system from taking any memory away from the virtual memory system. Thus the response time is the same regardless of the amount of time that the interactive program pauses between successive touching of its memory.

Surprisingly, the file system penalty actually improves the execution time of the *sort* benchmark relative to without the penalty (see Tables 3 and 4). When the file system is penalized, *sort* takes only 25% longer than the best case. This degradation is nearly identical to the degradation shown in [NWO88] when *sort* was run using only a small cache.

6.4. How Much of a Penalty?

The results of the benchmarks in this section show that penalizing the file system can improve interactive response without degrading overall system performance. In some cases, it can even make the performance of both file- and virtual-memory intensive programs better. However, it is not clear what the optimal penalty should be. The penalty should be large enough so that idle user programs that will be used in the near future will not be removed from memory, but not so large that the performance of the file system is degraded unnecessarily. The best value for the penalty will depend on the behavior of the users of the system. In Sprite we normally set the penalty to 20 minutes. This means that an interactive program's pages will not be reclaimed by the file cache until the program has been idle for 20 minutes.

7. Conclusions

Different size file caches are required for different program mixes. As a result, variable-size cache mechanisms are required to provide good performance for all types of programs. Unfortunately, standard variable-size cache mechanisms that treat virtual-memory and file data equally are not good enough. In order to ensure good interactive response to users, virtual memory data accessed in the last several minutes must be kept memory resident if possible - even if this requires removing more recently accessed file data. Basically, the file cache should be limited to those pages that are not required by the virtual memory system.

Fortunately, the Sprite variable-size cache mechanism has the ability to favor virtual memory pages. This is easy in Sprite because the virtual-memory system and the file system are kept separate. We use the Sprite file system penalty mechanism as we do our day to day work on Sprite. Since we began penalizing the file system, we have noticed that a file intensive program is no longer capable of ruining interactive response. It is my advice to implementors of other variable-size caching mechanisms (e.g. mapped files) that they include in their

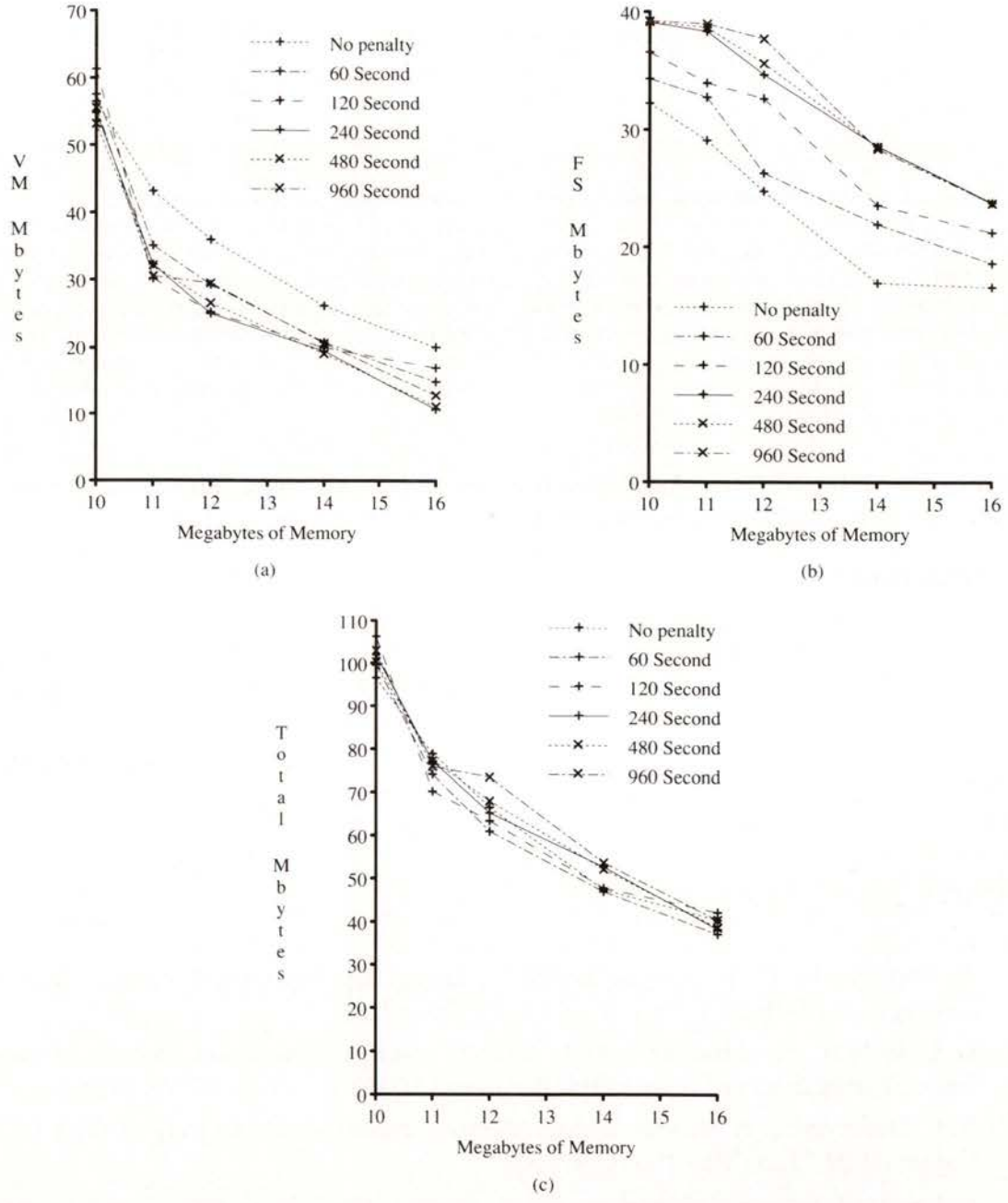


Figure 6. These graphs gives the number of Mbytes transferred across the network with various penalties as a function of client memory size. In all three graphs the X-axis is the total amount of client memory and the Y-axis is the total number of Mbytes transferred during the benchmark. Graph (a) is virtual memory traffic, (b) is file system traffic and (c) is total network traffic which includes virtual memory traffic, file system traffic and packet headers and control packets.

Sleep	Response Time			Sort Time		Page	Page	Cache Size	
Interval	Min	Max	Avg	Time	Deg	Ins	Outs	Min	Max
1	0.0	0.4	0.03	74.8	25%	1	4	64	178
5	0.0	0.1	0.02	72.8	21%	0	0	64	168
10	0.0	0.1	0.01	72.1	20%	0	0	64	168
30	0.0	0.1	0.03	74.0	23%	0	0	64	168

Table 4. Results for the IFS benchmark with variable-size caches when the file system is penalized 120 seconds. Each data point is the average of the results from three runs of the benchmark. The first column gives the number of seconds that the interactive benchmark slept before touching all of its memory. Columns 2 through 4 give the minimum, maximum and average number of seconds it took the interactive benchmark to touch all of its memory when it awoke from its sleep. Columns 5 and 6 give the total number of seconds it took to execute the *sort* benchmark, and the amount of degradation relative to the standalone case which took 60 seconds. Column 7 is the number of pages read in from swap files and Column 8 is the number of pages written to swap files. Columns 9 and 10 give the minimum and maximum amount of memory in the cache in Kbytes.

implementation the ability to favor virtual-memory pages. This will make interactive users happier and may even improve the performance of file intensive programs.

8. Acknowledgements

I want to thank the other Sprite developers: John Ousterhout, Brent Welch, Fred Douglis, and Andrew Cherenson. Without their efforts Sprite would not exist. John Ousterhout, Jeff Mogul, and Anita Borg provided numerous helpful comments that improved the presentation of this paper.

The work described here was done as part of my PhD research at the University of California at Berkeley. It was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00039-84-C-0107.

9. Bibliography

References

[BCD72] A. Bensoussan, C. T. Clingen and R. C. Daley, The MULTICS Virtual Memory: Concepts and Design, *Comm. ACM 15*, 5 (May 1972), .

[BBM72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, TENEX, a Paged Time Sharing System for the PDP-10, *Comm. ACM 15*, 3 (Mar. 1972), 1135-143.

[DaD68] R. C. Daley and J. B. Dennis, Virtual Memory, Processes and Sharing in MULTICS, *Comm. ACM 11*, 5 (May 1968), 306-312.

[LLH85] P. Leach, P. Levine, J. Hamilton and B. Stumpf, The File System of an Integrated Local Network, *Proc. of the 1985 ACM Computer Science Conference*, , Mar. 1985, 309-324.

[Lea83] P. J. Leach, et al., The Architecture of an Integrated Local Network, *IEEE Journal on Selected Areas in Communications SAC-1*, 5 (Nov. 1983), 842-857.

[Mur72] D. L. Murphy, Storage organization and management in TENEX, *Proceedings AFIPS Fall Joint Computer Conference 15*, 3 (1972), 23-32.

[Nel86] M. N. Nelson, The Sprite Virtual Memory System, Technical Report UCB/Computer Science Dept. 86/301, University of California, Berkeley, June

1986.

[Nel88] M. N. Nelson, *Physical Memory Management in a Network Operating System*, Phd Thesis, University of California at Berkeley, 1988.

[NWO88] M. N. Nelson, B. B. Welch and J. K. Ousterhout, Caching in the Sprite Network File System, *TOCS 6*, 1 (Feb. 1988), 134-154.

[OCD88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch, The Sprite Network Operating System, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

[RaR81] R. F. Rashid and G. G. Robertson, Accent: A communication oriented network operating system kernel, *Proceedings of the 8th Symposium on Operating Systems Principles*, , 1981, 164-175.

[RaF86] R. F. Rashid and R. Fitzgerald, The Integration of Virtual Memory Management and Interprocess Communication in Accent, *TOCS 4*, 2 (May 1986), 147-177.

[Ras87] R. Rashid, et al., Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, , Oct. 1987, 31-39.

[Red80] D. D. Redell, et al., Pilot: An Operating System for a Personal Computer, *Communications of the ACM 23*, 2 (Feb. 1980), 81-92.

WRL Research Reports

"Titan System Manual."
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

"Global Register Allocation at Link Time."
David W. Wall.
WRL Research Report 86/3, October 1986.

"Optimal Finned Heat Sinks."
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

"The Mahler Experience: Using an Intermediate Language as the Machine Description."
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

"The Packet Filter: An Efficient Mechanism for User-level Network Code."
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

"Fragmentation Considered Harmful."
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

"Cache Coherence in Distributed Systems."
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

"Register Windows vs. Register Allocation."
David W. Wall.
WRL Research Report 87/5, December 1987.

"Editing Graphical Objects Using Procedural Representations."
Paul J. Asente.
WRL Research Report 87/6, November 1987.

"The USENET Cookbook: an Experiment in Electronic Publication."
Brian K. Reid.
WRL Research Report 87/7, December 1987.

"MultiTitan: Four Architecture Papers."
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

"Fast Printed Circuit Board Routing."
Jeremy Dion.
WRL Research Report 88/1, March 1988.

"Compacting Garbage Collection with Ambiguous Roots."
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

"The Experimental Literature of The Internet: An Annotated Bibliography."
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

"Measured Capacity of an Ethernet: Myths and Reality."
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

"Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description."
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

"SCHEME->C A Portable Scheme-to-C Compiler."
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

"Optimal Group Distribution in Carry-Skip Adders."
Silvio Turrini.
WRL Research Report 89/2, February 1989.

"Precise Robotic Paste Dot Dispensing."
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

"Simple and Flexible Datagram Access Controls for Unix-based Gateways."
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

"Spritely NFS: Implementation and Performance of Cache-Consistency Protocols."
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

"Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines."
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

"A Unified Vector/Scalar Floating-Point Architecture."
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

"Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU."
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

"Integration and Packaging Plateaus of Processor Performance."
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

"A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance."
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

"The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance."
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

"Long Address Traces from RISC Machines: Generation and Analysis."
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

"Link-Time Code Modification."
David W. Wall.
WRL Research Report 89/17, September 1989.

"Noise Issues in the ECL Circuit Family."
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

"Efficient Generation of Test Patterns Using Boolean Satisfiablilty."
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

"Two Papers on Test Pattern Generation."
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

"Virtual Memory vs. The File System."
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

WRL Technical Notes

"TCP/IP PrintServer: Print Server Protocol."
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

"TCP/IP PrintServer: Server Architecture and Implementation."
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

"Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer."
Joel McCormack.
WRL Technical Note TN-9, September 1989.

"Why Aren't Operating Systems Getting Faster As Fast As Hardware?"
John Ousterhout.
WRL Technical Note TN-11, October 1989.

"Mostly-Copying Garbage Collection Picks Up Generations and C++."
Joel Bartlett.
WRL Technical Note TN-12, October 1989.

MTOOL: A Method For Detecting Memory Bottlenecks

Aaron Goldberg
Digital Equipment Corporation Western Research Laboratory

John Hennessy
Stanford University

December, 1990

Abstract

This paper presents a new method for detecting regions of a program where the memory hierarchy is performing poorly. By observing where actual measured execution time differs from the time predicted given a perfect memory system, we can isolate memory bottlenecks. MTOOL, an implementation of the approach aimed at Fortran programs running on MIPS-chip based workstations is described and results for some of the Perfect Club and SPEC benchmarks are reported.

This research was supported in part by an NSF Fellowship.

1 Introduction

Many modern computer architectures including cache-based uniprocessors and most shared memory multiprocessors present the programmer with a (deceptive) uniform access model of memory. RISC architectures for example provide simple load and store instructions to represent memory operations. In practice, however, the load instruction may involve accessing on-chip cache which is backed by a second level cache of static RAM which is backed by main memory. The time difference between a hit in the first level cache and a miss to main memory can be one to two orders of magnitude (see Table 1).

Programmers seeking to improve performance sometimes find it useful to optimize an algorithm with respect to a particular memory hierarchy. Studies like [1], [2], [7] and [8] have reported speed-ups of 100% and more owing to improved cache performance when nested loops are reordered and matrix algorithms are blocked. To make use of such techniques, the user must know where memory bottlenecks lie and when a transformation improves performance.

Two techniques are typically employed to isolate memory bottlenecks. The least time consuming approach is to statically analyze a program, using dependency analysis to identify how many items will be in cache after a certain number of iterations of a loop [3, 6]. Such techniques are important because they are potentially fast enough to incorporate into compilers to automatically manage transformations. However, the static techniques rely on the simple structure of both the loop and the memory system to perform their analyses. The approximate nature of analytic techniques and the in-

	Miss Penalty (cycles)		
Machine	Primary	Secondary	Remote
DEC 3100	6	–	–
DEC 5000	10	–	–
SGI (4 node)	14	40	–
MIPS 6280	2-4	50	–
Stanford DASH	14	28	103-136

Table 1: Memory Hierarchy Penalties

creasing complexity of the memory hierarchies they attempt to model make them inappropriate as a complete performance debugging solution.

At the opposite end of the spectrum, there are trace driven simulators which simulate the execution of every memory reference in a program. These simulations can model the entire memory hierarchy and produce access time estimates for every variable reference in a program. The obvious drawback of simulation is its cost; for large programs, simulation may be quite expensive. Also, it is non-trivial to correctly model a complicated memory hierarchy, particularly when multiprogramming or multiprocessing is involved.

Our technique strikes a balance between the expense of simulation and the inaccuracy of static analysis. Our key observation is that if we assume memory access time is uniform then, at least for simpler architectures, it is relatively cheap to correctly estimate the CPU execution time of a program. By comparing this uniform access model estimate with actual observed execution time, we can isolate regions in a program where the memory hierarchy performs poorly.

The next section discusses the method for estimating execution time assuming constant memory access time. Section 3 describes how to use the estimate to isolate memory bottlenecks. Section 4 presents a memory bottleneck tool implementation, MTOOL, which runs on the DECstation 3100 and 5000. Section 5 provides examples of MTOOL's user interface. Section 6 reports results when the tool is run on the Perfect Club benchmarks and scientific benchmarks in the SPEC set. Finally, section 7 gives some conclusions about the breadth of applicability of our technique and suggests directions for future research.

2 Estimating Execution Time

Consider a computer where all instruction scheduling is handled by software (i.e., no hardware interlocks) and where each instruction (including memory access instructions) has a known, fixed execution time. For such a computer, we can determine the execution time of a program given instruction execution counts using the formula,

$$\begin{aligned} \textit{execution time} = {} & \textit{(\# of times ith instruction executes)} * \\ & \textit{(time per execution of instruction i)} \end{aligned}$$

Let us try to apply a similar technique to a RISC architecture. First we divide the program into basic blocks. A basic block is a group of instructions with a unique entry point such that when the entry instruction executes, all other instructions in the basic block will execute. It is possible to identify all basic blocks in most executable programs by examining branch instruction destinations and indirect jump tables. After determining the basic blocks, we instrument the executable file by preceding each block with code to increment a counter. Running the instrumented program produces a table of basic block counts. A discussion of methods for instrumenting compiled code is provided in the Appendix.

Using the counts and our knowledge of when hardware interlocks are triggered, we can estimate how long each basic block executes. Our estimates will have two shortcomings:

1. Memory access instructions do not execute in constant time.

2. There may be hardware interlocks across basic block boundaries.

The first shortcoming is actually the feature on which our bottleneck detection technique is based. We assume all memory accesses take the minimum possible time (typically the time for a primary cache hit) and when our prediction disagrees with measured execution time we report a bottleneck.

The second weakness is not a problem for many RISC architectures because there are few hardware interlocks and these interlocks rarely cross basic block boundaries. On the MIPS processor where our experiments were performed, the inter-block interlocks were negligible in real code. If such interlocks occur with appreciable frequency, they can be estimated by instrumenting to collect branch frequencies as well as basic block counts. The branch frequencies tell us how often one basic block precedes another and we can improve our estimate by including interlocks between adjacent basic blocks.

Thus, we have a technique for estimating execution time of a whole program. Moreover, our method costs only one instrumented program execution rather than requiring a full, expensive machine simulation. The task now is to use this estimation technique to isolate bottlenecks.

3 Isolating Bottlenecks

In this section, we develop a framework for detecting bottlenecks by measuring divergence from predicted behavior. We begin by formalizing the notion of measuring actual time spent in a region of code. A measurable object or *m-object* is a set of instructions in which we can identify all entry and exit points. The object is measurable because we can place start timer and stop timer calls at these entry and exit points to measure the time spent in the object. For example, we can time a procedure by placing a *start_timer* call at the top of the procedure and *stop_timer* calls before every `return` statement. Similarly, we can time a loop by placing a *start_timer* above the top of the loop and a *stop_timer* below the bottom of the loop.

We say an m-object is *timable* if we can instrument the program to measure the time spent within the object. A timable object must satisfy two criteria:

1. The total time spent within the object substantially exceeds timer granularity.

2. The perturbation created by the timer is not significant.

The first aspect of timability is rarely a problem as most systems provide at least a 1/60th of a second timer, and m-objects of interest typically execute for seconds, minutes, or even hours. The perturbation issue is more difficult. To avoid changing memory performance, we require that the number of memory operations performed by the m-object substantially exceed the number performed in a clock timer call. In addition, to avoid appreciably slowing the program down, we require that the time spent in the m-object substantially exceed the time to make a clock call.

Using the above criteria, we can identify regions of the program whose actual execution times can be measured. These execution times include, however, not only the work done in an m-object proper, but also the work done on behalf of the object by any procedures that it calls. In contrast, the basic block counting estimation technique of the previous section calculates only the work done in a basic block; it ignores the time spent in procedure calls. Furthermore, while we can estimate the total time spent in a procedure q, we cannot necessarily determine the time spent in q on behalf of a particular caller. Thus, we cannot always estimate the time spent in and on behalf of an

m-object that calls q. We will say that the m-objects whose execution time can be accurately predicted by basic-block counting techniques are *estimable.*

To identify estimable m-objects, we exploit information about the structure of a program's call graph. In a call graph, nodes represent procedures and there is a directed edge from node v to node w if procedure v calls w during the execution of the program. The call graph has a distinguished node, the root, which is the procedure where execution begins.

We say a node v dominates w if every path through the graph from the root to w passes through v. The useful aspect of the call graph is that a node is estimable if it dominates all of its descendants. Intuitively, if a node v dominates w then all the work in w is done on behalf of v. Hence, if a node dominates all of its children, then the estimated time for that node and its descendants is just the sum of each of their estimated times, ignoring procedure calls.

This observation serves as a working definition of estimability. The definition implies that both the root of the call graph, which corresponds to the execution of the whole program, and the leaves which correspond to call-free procedures, are estimable. Given this operational definition of estimability, the primary issue in isolating memory bottlenecks now becomes one of granularity of detection.

We could estimate the execution time of the full program and compare this number against actual run time, but this will only describe the magnitude of the memory effects, not localize them. Instead, our approach is to find a set of smaller, timable m-objects containing the majority of memory operations, and then to select members of this set that are estimable. The next section outlines our algorithm.

4 The Implementation

This section describes one implementation of the estimable, timable m-object approach to isolating memory bottlenecks, MTOOL. This specific implementation is for Fortran programs running on MIPS-chip based workstations. Fortran was chosen as a target language because large Fortran programs often have memory bottlenecked regions and much of the research on alleviating memory bottlenecks has concentrated on scientific code.

MTOOL seeks to isolate bottlenecks at the level of procedures and loops.

This decision reflects the fact that procedures and loops have natural meaning to the user, satisfy the definition of m-object, and typically run long enough to meet the timability criteria. The first step of the bottleneck isolation process is to instrument the program of interest to collect basic block counts and to run the instrumented code on a representative input. The basic block counts are useful not only for estimating execution time; they also provide MTOOL with precise knowledge of where memory operations are concentrated.

MTOOL sorts the procedures by the number of memory operations they execute and selects those that contain the first 95% of all memory operations. For each of these procedures, MTOOL makes a list of loops and tries to measure first the individual loops and then the whole procedure, subject to timability and estimability constraints. Timability constraints are strongly system dependent. DEC's ULTRIX provides a 1/60th of a second granularity clock, but a system call is required to read the clock.

The overhead of the system call perturbs execution undesirably. The cleanest solution would be to modify the operating system to create a clock directly accessible by user processes, but a simpler, more widely applicable option was to add an interval timer to create a clock in user memory. Start and stop clock calls access the clock in user memory which is updated by the interrupts of the interval timer.

Using the user memory clock, MTOOL's timer has granularity of 1/60th of a second and work per start/stop call of about 70 instructions. The results reported in Section 6 seem to indicate this granularity and overhead are acceptable. Also, we expect that interest in characterizing and improving performance will drive architects and operating systems programmers to provide better clocks in the future.

At this point, MTOOL has gathered a collection of timable loops and procedures. The next step is to determine which of these objects is estimable. Conceptually, MTOOL uses a simple depth first algorithm to label each procedure in the call graph with two parameters:

$$Tot(v) = 1 + \sum_{calls\ from\ v\ to\ w} Tot(w) * \frac{\#\ of\ calls\ from\ v\ to\ w}{total\ \#\ of\ calls\ to\ w} \text{ and}$$

$$TDES(w) = 1 + number\ of\ descendants\ of\ w.$$

It is easy to show that for every node, $Tot(v) \leq TDES(v)$ and $Tot(v) =$

$TDES(v)$ exactly when v is estimable. This relation formalizes the observation of the previous section that a node is estimable when all the work of its descendants is done on its behalf.

Thus, we have a test for estimability. Extending the test to loops simply involves checking the analogous condition that:

$$\sum_{w \text{ called in loop}} TDES(w) + 1 = \sum_{w \text{ called in loop}} Tot(w) * \frac{\#\ of\ calls\ from\ loop\ to\ w}{total\ \#\ of\ calls\ to\ w}.$$

In most cases, the loops and procedures selected by MTOOL immediately satisfy the estimability condition. The condition is met frequently because the objects selected by MTOOL contain the majority of memory operations which typically means they involve the most frequently executed portions of code which are normally leaves or objects all of whose children are leaves.

When the test is not satisfied immediately, several options are available. One natural choice is to move up the call graph as we know that eventually we will encounter an estimable node, the root. This option is not ideal, however, because it reduces the precision with which we localize bottlenecks. A better choice is to check whether we can in fact accurately estimate the time spent in a procedure call. Below, we describe three cases where we can make an accurate estimate.

Many scientific library routines are simple, loop-free leaf procedures that always follow the same execution path. Their estimated execution times are consequently constant. We can often identify such procedures by checking that they meet two conditions:

1. The procedure is a loop-free and call-free.

2. The average time per call as determined by basic block counts is equal to the maximum or minimum possible time per call.

The first condition implies the procedure is estimable and that we can run shortest and longest path algorithms on the control flow graph of the procedure to bound its execution time. Using these bounds, we can check the second condition which implies that the execution time per call is constant because average equals extremum. This test finds that such common library calls as `SQRT()` and `EXP()` have constant execution times when called in the Perfect Club benchmarks.

1. Instrument executable program and collect basic block counts.
2. Select loops and procedures containing most memory operations.
3. Eliminate selected objects that fail to meet timability constraints.
4. Eliminate selected objects that fail to meet estimability constraints.
5. Instrument code to measure actual time spent in remaining selected objects.
6. Run instrumented code, correlate actual times with estimated times to isolate bottlenecks, and report bottlenecks to the user.

Figure 1: MTOOL's Bottleneck Isolation Algorithm

Two other simple heuristics suffice to eliminate many other problematic calls. Suppose object v calls procedure p. Then, we can normally assume average time per call from v to p is average time per call to v when either:

1. The vast majority (98%) of calls to p are made by v, or
2. (avg. time per call to p) * (# of calls from v to p) $\ll$ time spent in v, so any error in the approximation is negligible.

Both of these heuristics can be inaccurate under pathological conditions (when the variance of the execution time of p across calls is large), so MTOOL issues a warning whenever it invokes them. They have not caused problems with the benchmarks measured in this study.

The steps MTOOL uses to isolate memory bottlenecks are summarized in Figure 1. Step 6 is of course the most significant to the user.

5 User Interface

The user view of MTOOL is considerably less complex than the algorithms of the previous sections. The user types `MTOOL program-name input-files` and waits while MTOOL instruments the program to collect basic block

```
Predicted User Time: 95.0
Measured Time (compensated for counters): user 132.4    sys 0.7
Overhead estimates:       User (memory)        System (I/O)
                          39.3                   0.7
Proceed with memory bottleneck probe insertion (y/n)?
```

Figure 2: MTOOL's Initial Bottleneck Estimate

counts, runs and times the instrumented code, and displays a result like that shown in Figure 2. MTOOL generates the data in the figure by estimating the run-time of the program and comparing it with the measured execution time, appropriately adjusted for the effects of the counters. The information prevents a a user from proceeding when no significant memory bottlenecks are present.

Assuming the user asks MTOOL to produce an instrumented program, MTOOL executes steps 2 to 5 of Figure 1 producing a file of m-object descriptors and actual execution time measurements. This summary file is handed off to the front end.

The front end's top level window is shown in Figure 3 for the Perfect Club benchmark TFS.f, an air flow simulation. Overhead is defined as

$$(\textit{actual time} - \textit{estimated time})/\textit{estimated time}.$$

The histogram in the lower left of the window visually summarizes the data in the upper right. The line *"Measured 91% of memory operations"* reports the percentage of all executed memory operations that are included in measured objects. All times and overheads refer to user time only; time spent in the operating system on behalf of a program is ignored (though it was reported at an earlier stage–see Figure 2).

By pressing the histogram button, the user can obtain a histogram where bars represent total time spent on behalf of a procedure (Fig. 4). This time is split into estimated time and measured memory overhead. The bars are sorted in decreasing order of memory bottleneck magnitude. By clicking the mouse on the memory overhead portion of a bar, the user opens a text window (Fig. 5) displaying the procedure.

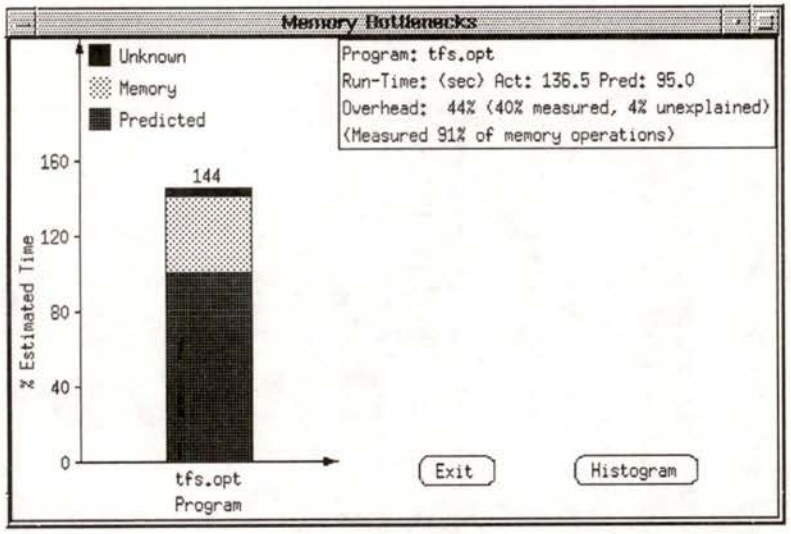


Figure 3: Top Level Window

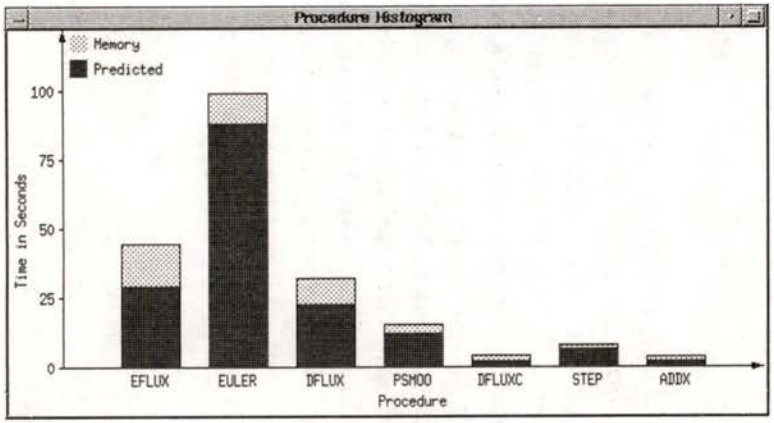


Figure 4: Procedure Histogram

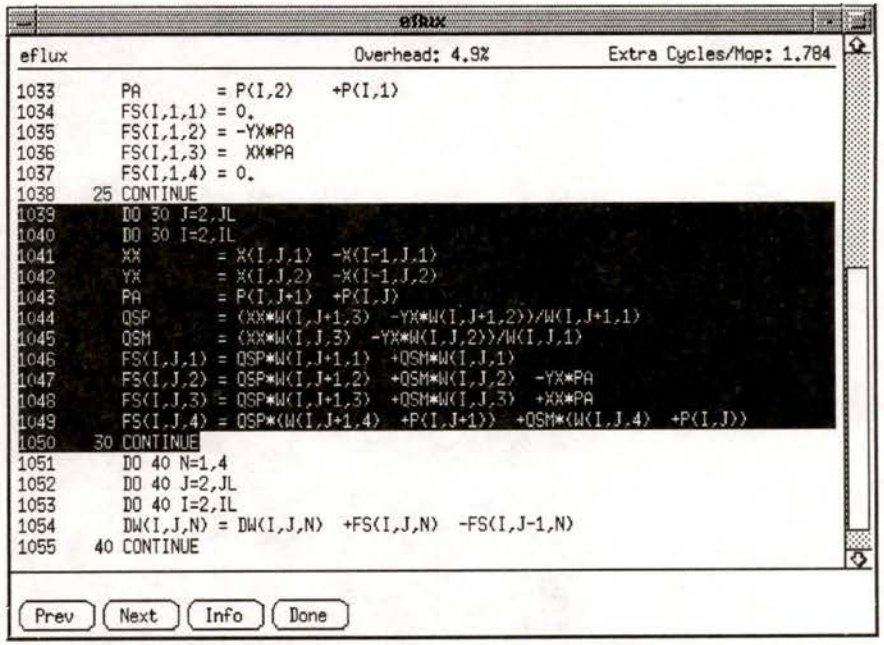


Figure 5: Displayed Bottleneck

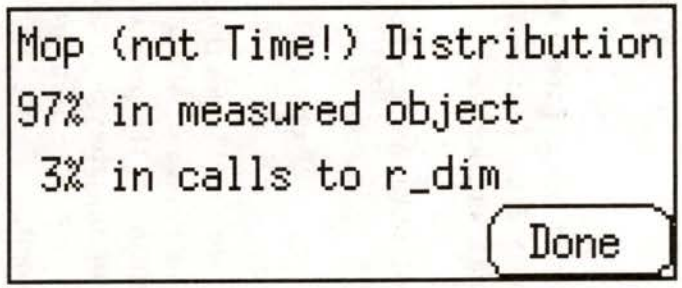


Figure 6: Info Pop-Up Window

Measured bottlenecks within the procedure are displayed by highlighting the text. Bottlenecks are highlighted one at a time, with the overall overhead contribution and the extra cycles per memory operation associated with the bottleneck reported at the top of the text window. Pressing the INFO button opens a popup window that gives further information about the distribution of memory operations when the measured object includes loops or procedure calls. Figure 6 shows a simple INFO window for the top bottleneck in `dflux()`. Pressing PREV or NEXT displays the previous or next bottleneck; bottlenecks are sorted by magnitude. The text window may be scrolled by pressing the vertical arrows at the right side of the window, and multiple text windows may be open simultaneously.

Three aspects of the user interface deserve comment from an implementor's perspective. First, the interface is relatively portable because it is built using Interviews [4], an object oriented toolkit that runs on top of X. Second, the notion of clicking on a bar to obtain further information on the associated bottleneck is quite general. For example, it would be easy to support an option where clicking on a procedure's estimated time bar would provide information implicit in our time estimate like register usage, floating point stalls, MFLOPS, most often executed lines, etc. A third note about the interface is that is uses standard line number information available in the object file to relate basic block level information back to source code lines. In this sense, the user interface is language-independent.

6 Results

The final measure of MTOOL and its user interface is how well they isolate memory bottlenecks. Table 2 summarizes our results for a subset of the Perfect Club benchmarks and the scientific benchmarks in the SPEC suite. The column entitled "% of Memory Operations" indicates that our heuristics for selecting timable, estimable m-objects succeed in choosing good potential bottlenecks, as we measure over 90% of memory operations in all but two cases where we measure 85%. The column on unexplained overhead, which reports the difference between measured overhead for the whole program and the sum of measured overheads for individual objects confirms this conclusion, as our measured m-objects typically account for all but a few percent of overhead. The least positive result is the data on number of source lines

Program	% of mops	Over-head	Unex-plained Overhd	# of Source Lines			
				In All m-obj.	Median/ m-obj	Max/ m-obj	In All Code
tfs	91%	40%	4%	252	8	53	2020
nas	97%	25%	0%	591	84	395	3976
sds	93%	6%	2%	167	12	120	7607
lws	93%	9%	0%	100	100	100	1237
lgs	98%	12%	0%	323	35	132	2327
fpppp	85%	50%	11%	940	274	666	2718
doduc	85%	29%	5%	2541	102	477	5334
spice2g6	95%	46%	2%	756	41	434	18411
dnasa7	97%	107%	1%	257	9	118	1105

Table 2: MTOOL Effectiveness

contained in the m-objects. On half of the programs, the median object contains fewer than 50 lines, but for some programs like fpppp, we are perhaps failing to localize bottlenecks adequately. This problem is discussed further in the Conclusions section below.

7 Conclusions

The results reported above support the general conclusion that our technique succeeds in detecting memory bottlenecks in scientific Fortran programs. One shortcoming of the current implementation is that it does not satisfactorily localize the bottlenecks in a few of the programs (see Table 2). This problem can certainly be addressed by modifying our algorithm to select smaller m-objects. Currently, MTOOL searches for timable, estimable m-objects starting with outer loops. Some benefit could be derived by trying inner loops first. Similarly, MTOOL will time a whole loop-free procedure, regardless of how many lines it contains. It would be a simple matter to add heuristics to partition large procedures into multiple m-objects. This new heuristic would benefit programs like fpppp in the SPEC set where a key bottleneck is a 700 line loop-free procedure. Finally, we could add a new option where MTOOL displays a procedure with its frequently executed memory references highlighted whenever the MTOOL selected m-object exceeds a certain line

count threshold. The user could then manually create acceptably proportioned m-objects containing these highlighted memory operations, subject to MTOOL's checks for estimability and timability.

MTOOL could also be enhanced by broadening the class of programs for which it can detect bottlenecks. MTOOL is now restricted to non-recursive programs that do not use procedure variables. The restriction on recursion derives primarily from the fact that the simple start/stop clock timers will not work when an m-object can be re-entered (and the clock re-started) before it is exited (and the clock is stopped). The restriction could be removed by prohibiting the timing of recursive procedures or by using more complicated timers that keep track of the depth of the recursive call and only start and stop the clock on the first entry and last exit. Because MTOOL is language independent (except for the restriction on recursion) modifying the timers (and modifying the check for estimability of section 4 to account for loops in the call graph) would allow MTOOL to handle languages with recursion.

The restriction on procedure variables is also easy to remove. MTOOL's definition of estimability reqires a well-defined call graph and the use of procedure variables means that the full structure of the call graph is not determined until run-time. By modifying the basic block counting instrumentation to record a dynamic call graph, this restriction could be circumvented.

The final and perhaps most exciting application for MTOOL technology is porting it to a shared memory multiprocessor. Memory bottlenecks on shared memory machines can be severe and detecting them is difficult. Analytic techniques cannot handle the complexity of parallel systems and simulating multiple interacting processors is extremely complex and expensive. MTOOL should provide a viable alternative to these approaches.

Acknowledgements: I would like to thank David Wall of DEC WRL who sponsored part of this research during a summer internship and provided invaluable advice on the ins and outs of patching code.

8 Appendix: Instrumenting Code

MTOOL requires the ability to modify an executable file in a non-intrusive manner. One approach used by the Pixie [5] program from MIPS is to directly patch an executable. A problem arises because some jumps are indirect; they

have the form `JR r1` where *r1* is a register containing an address. Pixie's solution is to include an indirect jump table which maps every address in the original executable to its corresponding address in the patched executable. Indirect jumps are always made through this table. The drawback of Pixie is that it can introduce non-negligible overhead.

A second approach is to patch the code at link time, modifying pc-relative jumps, text addresses in the data segment, and the relocation dictionary to correctly reflect changes in the executable file. The advantage of this approach is that all jump destination addresses are clearly identifiable and no overhead is added. See [9] for a a complete description of the technique.

MTOOL uses a third approach. MTOOL patches the executable directly, but it does not use an indirect jump table. Instead, MTOOL pattern matches to find instructions which load addresses in the text segment. The loaded text address is modified to represent the corresponding address in the instrumented code. Thus, a `JR r1` will succeed because the instructions to load *r1* with a value have been correctly updated. This technique suffers from the potential drawback that the pattern matcher could fail in highly optimized code, but we have encountered no problems to date.

References

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transanctions on Computing*, 30(5):341–56, 1981.

[2] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. Technical Report CSRD 625, University of Illinois, 1987.

[3] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, 1988.

[4] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, 1989.

[5] Mips Computer Systems Inc., Sunnyvale, CA. *Language Programmer's Guide*, 1986.

[6] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989.

[7] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations – exploiting the memory hierarchy. To appear *ACM Transactions on Mathematical Software.*

[8] E. Rothberg and A. Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. To appear *Proceedings of Supercomputing '90*, Nov. 1990.

[9] D. Wall. Link-time code generation. Technical Report 89/17, DEC Western Research Lab, 1989.

WRL Research Reports

"Titan System Manual."
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

"Global Register Allocation at Link Time."
David W. Wall.
WRL Research Report 86/3, October 1986.

"Optimal Finned Heat Sinks."
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

"The Mahler Experience: Using an Intermediate Language as the Machine Description."
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

"The Packet Filter: An Efficient Mechanism for User-level Network Code."
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

"Fragmentation Considered Harmful."
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

"Cache Coherence in Distributed Systems."
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

"Register Windows vs. Register Allocation."
David W. Wall.
WRL Research Report 87/5, December 1987.

"Editing Graphical Objects Using Procedural Representations."
Paul J. Asente.
WRL Research Report 87/6, November 1987.

"The USENET Cookbook: an Experiment in Electronic Publication."
Brian K. Reid.
WRL Research Report 87/7, December 1987.

"MultiTitan: Four Architecture Papers."
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

"Fast Printed Circuit Board Routing."
Jeremy Dion.
WRL Research Report 88/1, March 1988.

"Compacting Garbage Collection with Ambiguous Roots."
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

"The Experimental Literature of The Internet: An Annotated Bibliography."
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

"Measured Capacity of an Ethernet: Myths and Reality."
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

"Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description."
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

"SCHEME->C A Portable Scheme-to-C Compiler."
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

"Optimal Group Distribution in Carry-Skip Adders."
Silvio Turrini.
WRL Research Report 89/2, February 1989.

"Precise Robotic Paste Dot Dispensing."
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

"Simple and Flexible Datagram Access Controls for Unix-based Gateways."
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

"Spritely NFS: Implementation and Performance of Cache-Consistency Protocols."
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

"Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines."
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

"A Unified Vector/Scalar Floating-Point Architecture."
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

"Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU."
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

"Integration and Packaging Plateaus of Processor Performance."
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

"A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance."
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

"The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance."
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

"Long Address Traces from RISC Machines: Generation and Analysis."
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

"Link-Time Code Modification."
David W. Wall.
WRL Research Report 89/17, September 1989.

"Noise Issues in the ECL Circuit Family."
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

"Efficient Generation of Test Patterns Using Boolean Satisfiablilty."
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

"Two Papers on Test Pattern Generation."
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

"Virtual Memory vs. The File System."
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

"Efficient Use of Workstations for Passive Monitoring of Local Area Networks."
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

"A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units."
John S. Fitch.
WRL Research Report 90/6, July 1990.

"1990 DECWRL/Livermore Magic Release."
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

WRL Technical Notes

"TCP/IP PrintServer: Print Server Protocol."
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

"TCP/IP PrintServer: Server Architecture and Implementation."
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

"Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer."
Joel McCormack.
WRL Technical Note TN-9, September 1989.

"Why Aren't Operating Systems Getting Faster As Fast As Hardware?"
John Ousterhout.
WRL Technical Note TN-11, October 1989.

"Mostly-Copying Garbage Collection Picks Up Generations and C++."
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

"Limits of Instruction-Level Parallelism."
David W. Wall.
WRL Technical Note TN-15, December 1990.

"The Effect of Context Switches on Cache Performance."
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

"MTOOL: A Method For Detecting Memory Bottlenecks."
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

Trace Partitioned Parallel Cache Simulation

Viranjit S. Madan

Computer Sciences Department

University of Wisconsin - Madison

Outline

Why evaluate caches?

What is trace-driven simulation?

Parallelizing cache simulation

Experimental evaluation

Conclusions

Why Evaluate Caches?

Memory systems set sustained performance
(especially with super-microprocessors)

Caches

- low latency
- high bandwidth
- low cost

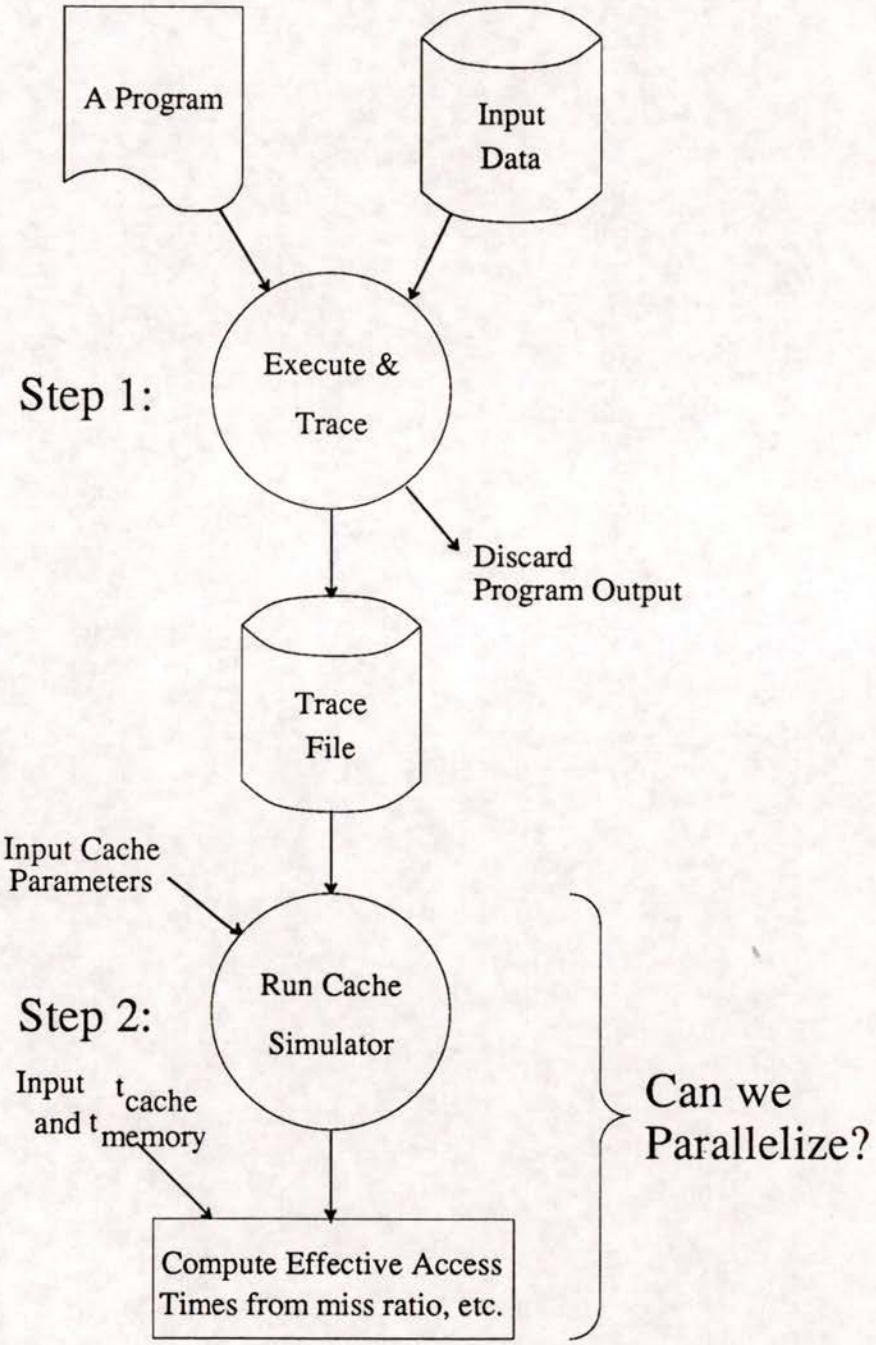
Re-evaluate caches

- workloads change
- technology changes

Trace-driven simulation is the preferred method

But it is time-consuming!

What is trace-driven cache simulation?



Parallelization Alternatives

Evaluate more than one cache at a time

[Mattson 1970] Stack Simulation

Fixed block size, fixed #sets, "inclusion" replacement

evaluate range of associativities (sizes)

[Hill & Smith 1989] All-Associativity Simulation

Fixed block size, LRU replacement

evaluate range of #sets and associativities

By parallel, *we* mean concurrent in time

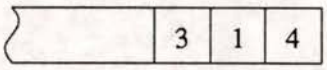
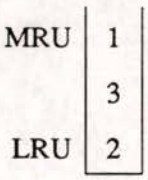
Use all-associativity alg. as starting point
Parallelize all-associativity

Fully-associative (FA) cache example for ease of presentation

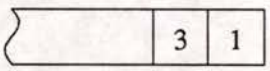
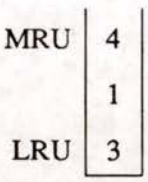
Implementation and results for general case

Cache simulation example

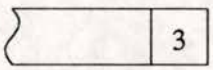
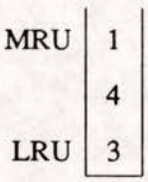
Assume a 3-block FA cache using LRU

TRACE	CACHE	METRICS
		Miss 12 Hit 20

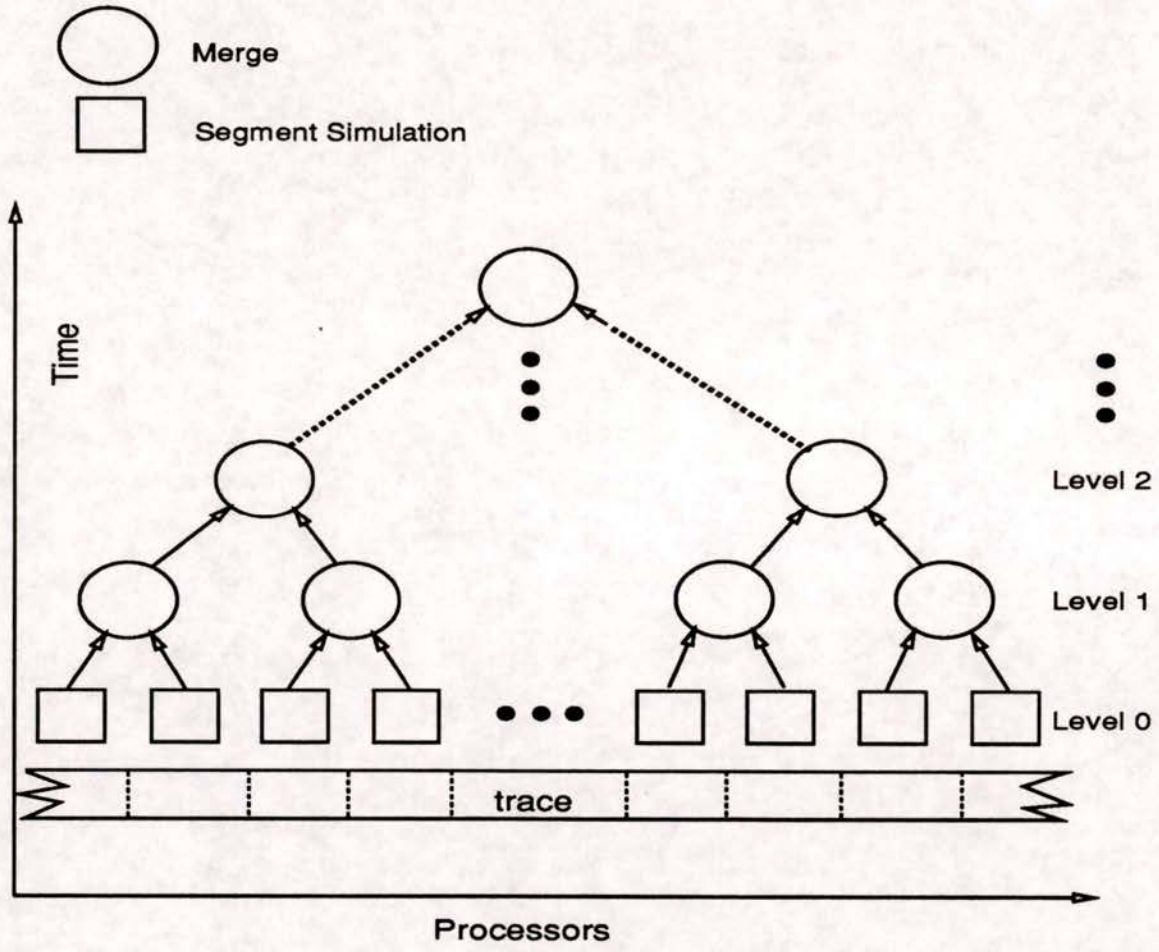
Processing reference 4 yields:

		Miss 13 Hit 20

Processing reference 1 yields:

		Miss 13 Hit 21

Appears ''inherently'' sequential


Merge
Segment Simulation
Time
Level 2
Level 1
Level 0
trace
Processors

Parallelizing cache simulation

Simulation of caches seems sequential, but

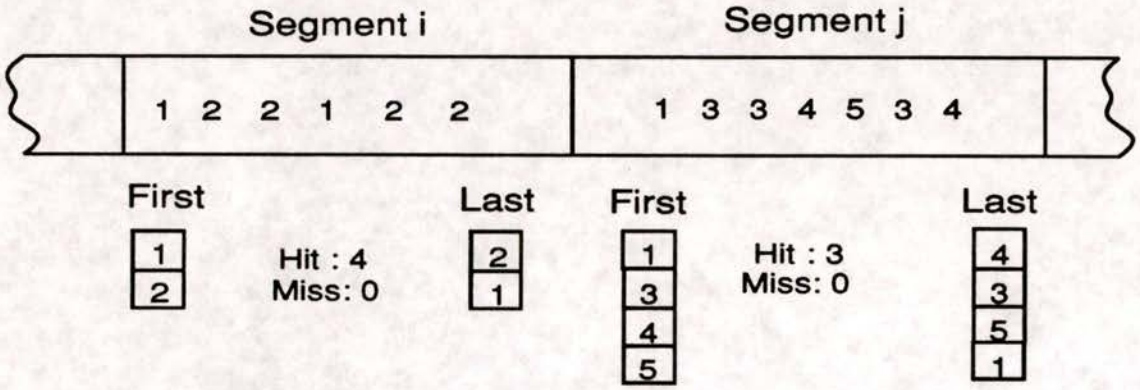
- Don't think of trace serial input
- Rather operate on whole trace

Approach of algorithm:

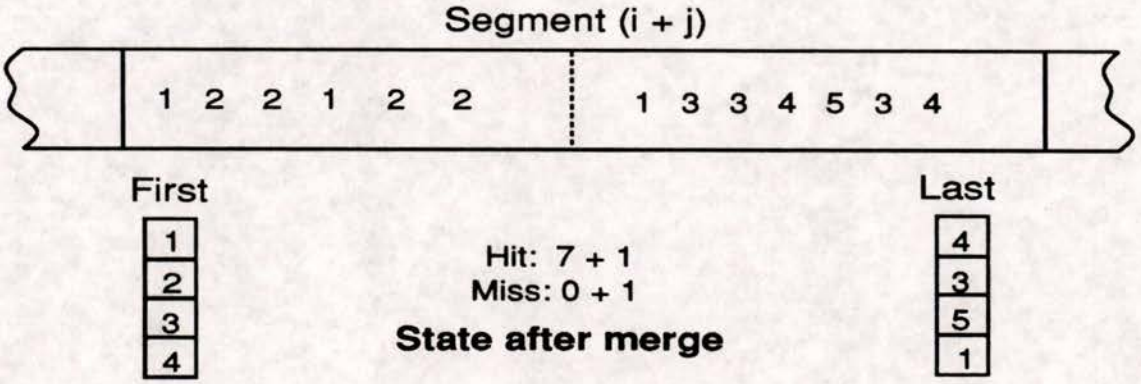
- Break trace into segments of consecutive references
- In Parallel
  - Simulate each segment
  - (Record history-dependent references & final state)
- In a Tree
  - Merge results from adjacent segments
  - (Merge associative)

Merge Example

Assume a 4–block FA cache using LRU



States before merge



State after merge

An Experimental Evaluation

DEC Western Research Lab traces

- Five traces from DEC Titan (a RISC)
- About 100M references each

All-Associativity Simulation

- Fixed block size (32 bytes)
- Varying associativities (1 to 8-way)
- Varying number of sets (cache sizes)

Test Cases

- small -- 32 to 256 sets (1KB to 64KB)
- large -- 512 to 4K sets (16KB to 1MB)
- small & large -- 32 to 4K sets

Implementation Particulars

Prototype

Conservative coding

Synchronization via files

Transfer of data using files

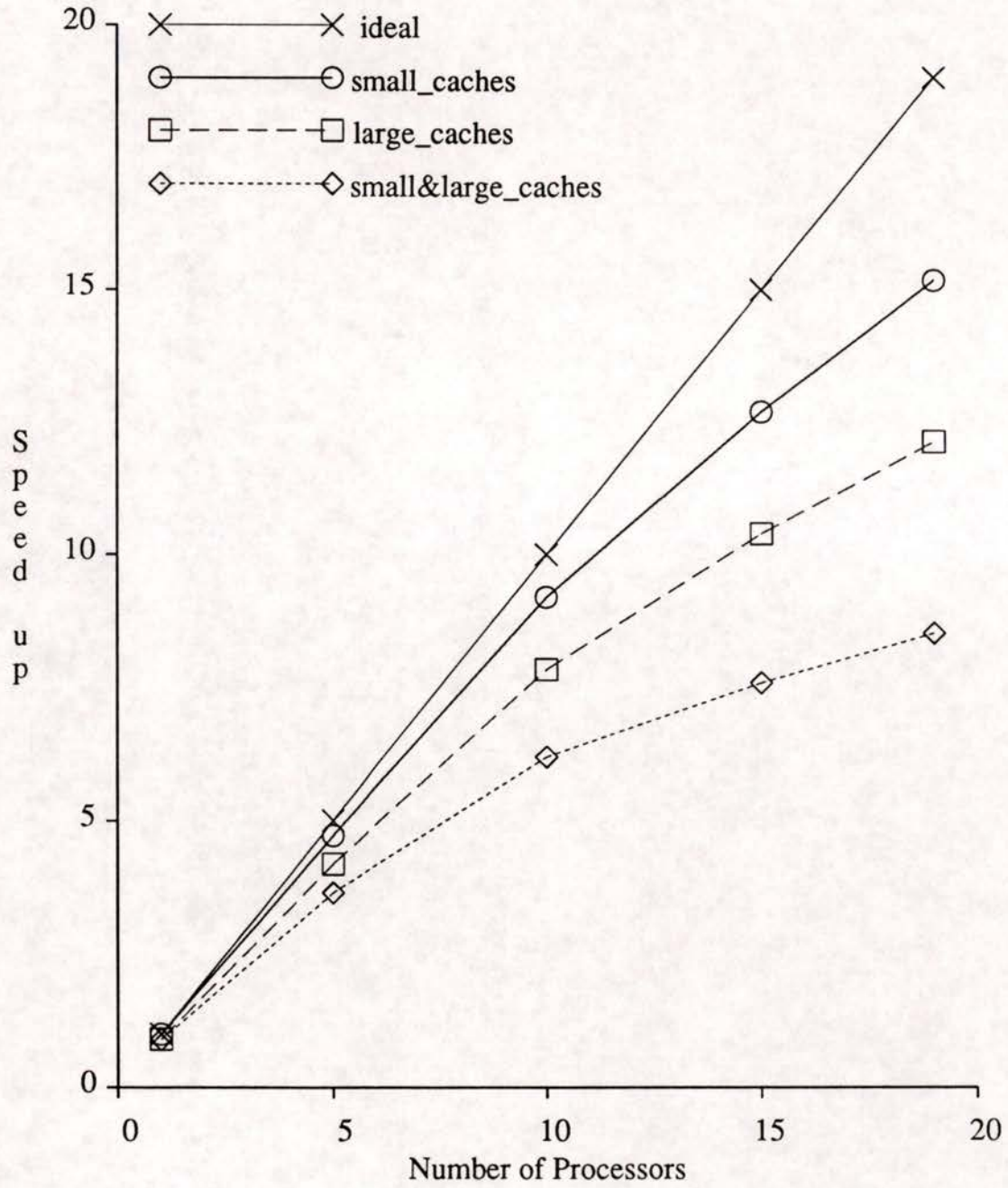
All files remote mounted (NFS)

$$Speedup = \frac{time_{serial}}{time_{parallel}}$$

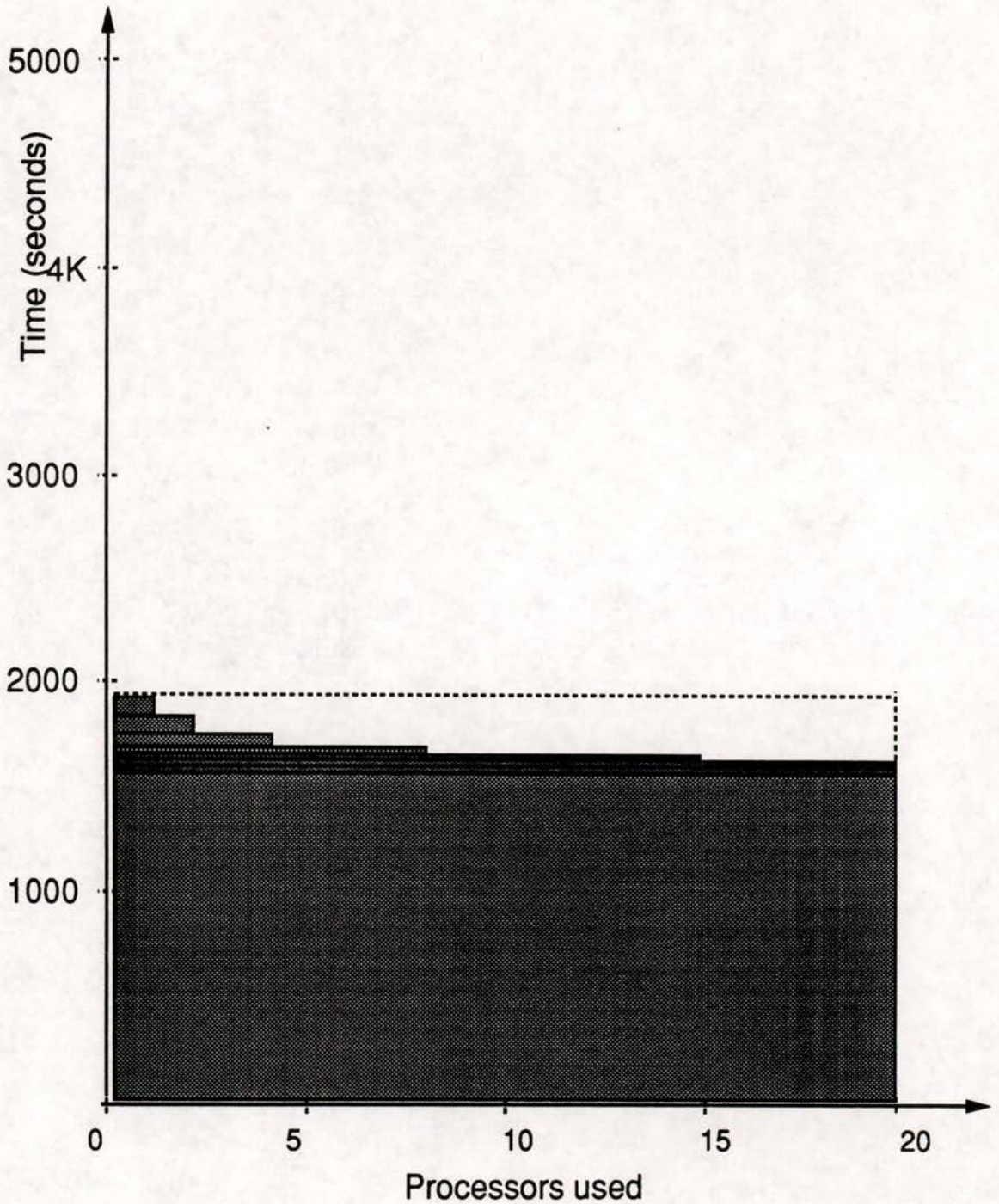
Serial version was best available

Therefore, results are conservative

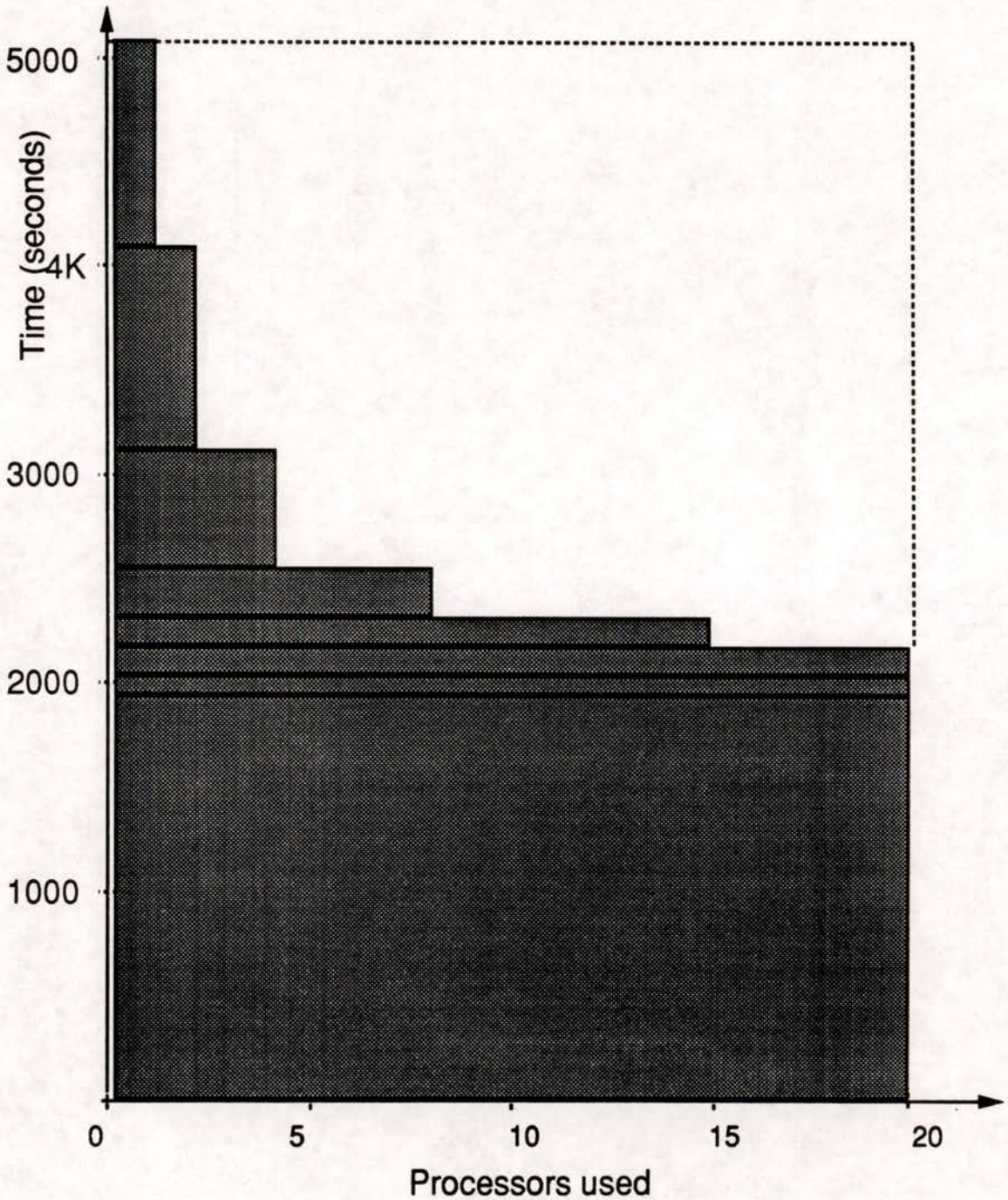
Speedups from Sequent Symmetry



Merge times for Large caches



Merge times for Small&Large caches



Analysis of Speedup (contd.)

Simulation of small & large caches gives poor speedup because

- Merges near root of merge tree take long
- Available parallelism is low near root of tree
- Extra processors cannot do useful work

Summary

Parallelizing trace driven cache simulation

- Break trace into segments
- Simulate each independently
- Merge results hierarchically

Experimental evaluation shows significant speedup

Further improvements need increase of available parallelism near root of merge tree

General approach may be useful in other simulation contexts.

- (example of Chandy-Misra space-time simulation)

Measuring Process Migration Effects Using an MP Simulator

Andrew Ladd, Trevor Mudge, and Oyekunle Olukotun
Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan

Abstract

This chapter examines multiprocessors that are used in throughput mode to multiprogram a number of unrelated sequential programs. This is the most common use for multiprocessors. There is no data sharing between the programs. Each program forms a process that is scheduled to execute on a processor until it blocks because its time-slice expires or for I/O. The process can migrate among the processors when its turn to resume occurs. The degree of process migration can dramatically impact the behavior of caches and hence the throughput of the multiprocessor. This chapter investigates the behavior of cache misses that can result from different degrees of process migration. The degree of migration is varied by assigning each process an affinity for a particular processor. An efficient multiprocessor cache simulator is described that is used in the study. The study is restricted to shared-bus multiprocessors and two contrasting cache consistency protocols, write update and invalidate.

INTRODUCTION

In today's computing environment, multiprocessors are playing a greater role than ever before. Originally, multiprocessors were intended to accelerate single program execution through parallel processing. However, their most common use to date has been as "throughput" machines in multiprogramming environments. These systems typically consist of 2 to 32 processors connected by a common backplane bus. Memory units are also attached to this backplane such that all processors have equal access to memory. This is commonly referred to as a tightly-coupled shared-memory system.

Programs are selected to execute on the processors of the machine by the operating system scheduler. In a typical case, the scheduler takes a program (process) from a common run queue and sends it to one of the available processors. When the process' time-slice expires or it blocks on a system call (for example when waiting on I/O), it is removed from its processor and another process from the run queue takes its place. This is characteristic of a preemptive scheduler. When the process is ready to execute again it is placed back on the run queue and waits its turn for execution. In a straightforward round-robin type of scheduling, there is no guarantee which processor this process will execute on next. With respect to cache performance, the best processor for it to execute on next is most likely the one it executed on last. There is a much greater likelihood that the process will find more of its initial memory references in the cache of the processor it executed on last, thereby reducing the number of instructions and data that must initially come from memory[1]. This gives the process an affinity for the cache of the processor it last left [2]. On the other hand, if the process is scheduled on a processor it has never executed on, it will have no footprint (matching data and instruction lines) in that processor's cache. In this case it will have to bring in all the instructions and data it initially needs from memory, making an initially cold cache warm to the process [3]. Bouncing a process from processor to processor in this manner is termed process migration and can have a detrimental affect on the overall cache performance and bus traffic in a computer system. Process migration also increases the

interference of a migrating process' address references with the cache footprints of other processes. This interference will displace cache lines from previously executed processes, thus reducing the cache footprint of those processes.

An understanding of these process migration effects on system performance is important for the operating system designer. If better scheduling algorithms can send a process to a warm cache (one that the process left recently) then the cache hit rate is improved and bus traffic is lowered during the initial execution of the process. Transient or compulsory misses, initial misses that result from warming up a cache to a process, are reduced if the process is scheduled to a cache which is already warm to it. Steady state misses are cache misses that result from memory references displacing valid cache lines of the executing process [4]. Process migration tends to increase the transient miss behavior of caches in the system while leaving the steady state miss behavior unaffected. Other aspects of the design can also benefit from understanding process migration effects. For instance, cache and bus designers should consider the impact of how process migration effects could degrade system behavior.

A method to keep the caches consistent is required because processes migrate from processor to processor. This is especially important when actual data sharing occurs between processes because each processor needs to have equal access to the shared data. In tightly-coupled shared-memory systems there are two basic types of consistency algorithms: invalidate and update based. The invalidate protocol is based on eliminating shared data in other processor's caches when one processor modifies its copy of that data. The update protocol updates shared data in other caches upon each modification of the data in the expectation that it will be reused.

The choice between using physical caches or virtual caches has many tradeoffs. Virtual cacheing does not require the translation of the virtual address in the TLB (Translation Lookaside Buffer) before accessing the virtual cache. This significantly reduces the cache access time after the generation of the virtual address [3]. Cache line synonyms can cause problems in a virtual cache. Synonyms occur when two cache lines indexed by different virtual addresses refer to the same physical address.

A problem arises when a processor tries to modify one of the synonym lines because it is difficult to identify the other synonyms for the required modification. Virtual caches also create added complexity in the bus watching of the consistency protocols. A reverse TLB is required to map physical bus addresses into virtual addresses because the references on the system bus are physically addressed.

The trend in cache design for processors has been to increase cache size as processor speeds have increased. This is to compensate for the growing gap between main memory speeds and processor speeds. Main memory cycle times (and intrinsic bus delays) have not decreased at the pace of processor cycle times. This makes memory references that miss the local caches even more costly. The easiest method to reduce this cost is to make the local caches larger. In most designs the first level cache (primary cache) is limited to the amount of chip area the CPU designers can set aside. One solution is to create a secondary level of cacheing that can alleviate the high cost of references missing the first level cache. The secondary cache is usually made up of standard SRAM parts assembled on the board in close proximity to the CPU and is typically slower than the primary cache. A secondary cache built this way is usually an order of magnitude larger than the primary cache. It is for these large cacheing systems that longer address tracing is crucial. Effectively analyzing these large caches requires a large address stream to measure its steady state behavior [5]. Address streams that are too short may only measure the initial warming of the cache (i.e., transient behavior).

The experiments conducted in this study examine cacheing behavior in a multiprocessing environment. Only process migration effects are considered, therefore processes do not share data between other processes. This restriction was made to simplify the tools and to focus on "throughput" environments. The tools used include an address trace generator and a multi-cache simulator. These tools provide fast and accurate measurements of cacheing behavior with respect to scheduling algorithms, consistency algorithms, and cache characteristics.

Method	references	slow-down	abstraction
Analytical	>1M	-	Yes
Trace Driven	>1M	10X	No
RTL	~10K	1000X	No

Table 1: Comparisons of Cache Analysis Methods.

Previous Studies

Many studies have focused on the sharing between parallel processes. These studies [6, 7, 8, 9, 10] are interested, primarily, in exploring the effects of cache consistency or coherency where there is shared data. Although some of these studies account for process migration, the central theme they explore is somewhat different from characterizing process migration effects on multiprocessor caches.

Some of these studies [6, 7, 8] use analytical models to simulate address streams. While they can produce results very quickly and have the most flexibility, their results are an abstraction of real system behavior. At the other end of the spectrum are studies [10] which completely simulate each processor in some form of detailed register transfer level. While this method produces the most accurate results, it is far too slow to capture large multiprogrammed address traces. Trace driven simulations offer a compromise between the two. They take real address traces from targeted programs and run them through a high level cache simulator. Trace driven methods are considerably faster than detailed models but still simulate "real" system behavior. Faster methods are becoming increasingly desirable because cache sizes are growing. As cache sizes grow, the need for longer traces is crucial to accurately measure the behavior of these large caches. Table 1 illustrates a comparison of these methods.

This study uses a high level cache simulator driven by address traces extracted from benchmark programs. The size of the traces excluded register transfer simulation from consideration due to the time required for RTL simulations. Analytical models were not considered due to their

inaccuracies.

SCHEDULING

A key aspect to examining process migration effects is to set up a multiprogramming model that is sufficiently realistic yet deterministic enough to allow the resimulation of the same sequence of scheduling events through various cache configurations. When studying migration effects, it is important to identify which address stream is produced by which process (program). Without this identification it is difficult to isolate cacheing behavior affected by process migration.

Not only is process identification important but so is finding an accurate algorithm to interleave the multiple processes (scheduling). One approach is to let the host system schedule the multiple programs with its native scheduler and have each program send its results to the same cache simulator. The simulator then processes the address streams as they are received. In this way, the cache simulator analyzes the address traces generated in a manner consistent with the host machine's scheduler. This provides a straightforward method to utilize a real scheduling algorithm [5]. One drawback to having scheduling performed by the host is that it is sensitive to traced code side-effects, since the host scheduler cannot account for the annotated code. Another drawback is that the traced code would rarely schedule the same way between separate executions due to the uncertainty of system loading.

An alternative scheduling approach is to take a known scheduling algorithm, or statistical measurement of one, and apply that directly to the programs which are executed. This approach can compensate for some of the tracing side-effects by altering the algorithm appropriately. It also allows repeatable scheduling for different executions of the programs.

Closely related to program interleaving is processor scheduling. As with interleaving, an algorithm is needed to schedule each processor in the multiprocessor set. This problem is less severe since a new context can start execution on any idle processor or the first that becomes available. Since measuring the effects of process migration and affinity is the topic of this chapter, the scheduler must be flexible enough to study various scheduling algorithms and their impact on system performance.

CACHE COHERENCY

A method to keep the caches consistent is required because processes migrate from processor to processor. This is especially important when actual data sharing occurs between processes. Since this study does not examine data sharing, why should cache coherency matter at all? To answer this question, one only has to note that the context of a process must be consistent with past contexts of that process. For example: if process A was on processor 1 and referenced location X, then location X is valid in that processor's cache. Now assume process A is swapped out of processor 1 and is later scheduled on processor 2. While executing on processor 2 it writes to location X. Without a consistency protocol, processor 1's cache line for X will not be updated or invalidated. Now if process A is re-scheduled back to processor 1 and reads X it will receive a stale value.

Since cache coherency is an important consideration with process migration, valuable insight can be gained by measuring different coherency effects on caches. Cache coherency algorithms have been studied at great length. Many papers have been devoted to the comparison of coherency based on directory schemes and snoopy protocols [6, 7, 8, 9, 10]. Although these studies focus primarily upon the sharing of data, valuable conclusions are drawn as to when each scheme is beneficial over the others. Directory schemes are ideally suited for a distributed bus type of organization while snoopy schemes are more suited for centralized shared bus organizations. Distributed bus organizations, and thus directory based coherency, are more applicable with large numbers of processors where a centralized bus cannot support the necessary bandwidth for the communication traffic between processors and memory. Centralized buses on the other hand are simpler and less costly to implement and are therefore suited for a modest number of processors. Although the directory mechanisms tend to reduce the bus traffic bottleneck they usually cause a bottleneck at the memory modules because memory must maintain the coherency tables. Both the directory and snoopy based schemes can support invalidation or update protocols.

Finally, software approaches keep caches coherent without causing bus

or memory bottlenecks. This is done by flushing a processor's cache at each context switch. Their primary drawback is the reduction of usable cached data because a process is guaranteed to enter a cold cache at each context switch.

ADDRESS TRACE GENERATION

Before a cache analysis of "real" multiprocessing system can begin, accurate address traces must be generated. Ideally the address trace contains the complete set of address streams as seen by each processor in a multiprocessor system, or the equivalent information. This allows a cache simulator to process addresses exactly as they occurred. There are many factors that prevent the generation of an ideal set of address traces. Such factors include the side-effects of generating or extracting the traces, the performance degradation between the traced code and the actual code, and the difficulties of tracing certain kernel routines.

One major deviation from the ideal is that in many instances one cannot acquire address trace data from all instructions executing on a particular CPU. The particular method used may restrict tracing only to user programs and ignore the tracing of kernel code. This is particularly burdensome when trying to measure how context switching affects caches since various kernel procedures are likely to execute repeatedly on each processor.

Another deviation is the intrusion of the tracing method on the traced program. Tracing is usually done by inserting code between load, store, and program control instructions to record memory references. The program control instructions provide basic block boundaries which identify program flow changes. Whether the code is inserted into microcode [9] or inserted into program code [5, 11, 12], it still only executes when a program is traced. This extra code obviously creates side-effects and behavior that is never seen in the original untraced code. Memory management side-effects are examples of abnormal behavior which manifest themselves from the added code. The added calls to the inserted code to record addresses could cause page faults and displace TLB entries not normally seen by the original program. In addition, the address traces must be written somewhere, whether to a UNIX^{TM} pipe or a file, and are

bound to cause page faulting and TLB displacement. Reducing most of these side-effects and abnormal behavior is another concern in selecting a good trace generation method.

Closely related to the side-effect issue is the performance degradation of the code that is traced. If additional code is added to the original program in order to trace the address references, there is little chance that this added code will not degrade program performance. Performance degradation can cause several unfortunate by-products. First, system clocks and timers appear to run faster than as seen by the original program. Since fewer instructions of the original program are executing between timed events it appears to the executing program that clocks and timers run faster. Furthermore, I/O will appear faster, the program will appear to have a smaller time-slice before preemption, and kernel interactions will appear to execute faster if kernel code is not traced. For example, the Titan address generation scheme [5] slows down the execution of a program by as much as 12 times. ATUM [9] increases the time of execution by a factor of 10 and AE [11] induces a factor of 4 increase in execution time.

Finally, an enormous amount of data is created when generating address traces for large runs. Manipulating this data into a manageable form is our last consideration in tracing data. Two approaches are available to solve this problem. The first and most straightforward approach [5, 12] pipes the traces from the traced file directly into the program that analyzes them. This saves all the overhead of storing the traces into a file and, depending upon the disk cacheing method, could save time. In addition, disk space is no longer an issue when using pipes. The primary drawback with this method is that not all experiments are repeatable with the same set of traces, especially those involving kernel interaction. Each run will produce a slightly different behavior which could skew results when comparing different cache configurations to the same programs. The second approach is to save the traces in compressed form [13], or to save a smaller set of events that can be used to extract the real traces [11]. These approaches allow different experiments to run on the same set of data but suffer from the extra file system overhead that the piped version eliminated. They also require a great deal of file space.

The Mache compaction scheme used with UNIX^{TM} compress has been shown to reduce the traced data to approximately 2.5 bits per reference [13]. The AE method of saving compressed significant events have been shown to reduce the traced data close to 0.5 bits per reference [11].

TOOLS

For this study, an address tracing tool was employed which annotates the object code of the executable program. This choice was made because it could generate addresses from the original program as well as the library routines that were called (since the annotation is performed after linking). The address tracer also allowed the address stream to be sent to a UNIX^{TM} file descriptor which simplified piping multiple streams into the cache simulator.

The cache simulator reads in the address streams from the various executing programs via the UNIX^{TM} file descriptors. In addition, the simulator can sequence the reading of the file descriptors according to any particular process scheduling algorithm. As the addresses were read, the cache arrays were examined and updated in accordance with the cache consistency algorithms. In this study, only snoopy bus algorithms were examined. Both update and invalidate protocols were modeled to study their respective effects on cache behavior in the presence of process migration.

The caches in the simulator are direct-mapped with variable index and line sizes. Tag matches are based upon the non-index and non-line portion of the physical address and the pid (process identifier) of the process. Cache lines are allocated on read, as well as write, misses. Context switching is modeled by scheduling a process to execute for 3000 instructions on average[14]. A Poisson distribution with an expected value of 3000 is generated when a process is scheduled to execute. This value is set as the limit to how long the context of the current process will run. Context switch times thereby distribute around a Poisson mean of 3000 instructions.

The pid of each process is included in each address tag. This allows the disambiguation of virtual addresses of different processes. Since no sharing between processes exist this modified tag causes no unwanted

mismatches between different process' addresses. Since only virtual address traces were available, special care was required to distribute the references in the cache. In a physical cache, virtual pages are translated into physical pages which create a random distribution over each process' address space. A method to emulate this behavior is highly desirable. Sites and Agarwal proposed a method to hash the virtual addresses based on the pid of the process[15]. They found that caches fed with hashed virtual addresses provided similar performance to physically addressed caches. Virtual address hashing gives a pseudo-random distribution of each process' virtual address space inside the physical cache. If no adjustment to the virtual addresses were made the miss rate would be unrealistically high because many programs reside in the same range of virtual space. Physical page allocation tends to randomize pages in this space thus causing fewer cache line conflicts between processes.

A piping program linked the address trace generation program to the cache simulator in such a way that the cache simulator could read off each trace from a unique file descriptor. The piping program created a UNIX^{TM} pipe for each executing program and sequenced the file descriptor for each pipe starting at a predetermined base.

Scheduling Details

Two types of scheduling are examined, one that schedules globally and one that schedules locally. The global scheduler takes the processor which has been idle the longest, or if none, the first to become available, and schedules the process that has been waiting the longest to execute. Jobs are maintained in queues that hold processes which are waiting for execution. As a process' time-slice expires, it is placed at the rear of the queue and the process at the head is then chosen to run on the first available processor. Process priorities are not considered in the scheduling mechanism, so all processes have equal priority and equal access to each processor. In contrast, the local scheduler assigns each process to a unique processor corresponding to its initial context of execution. From then on that process only executes on its assigned processor. While this local scheduler is highly unrealistic, it does show how complete process affinity can affect cacheing behavior. For this reason it was chosen for

this study. Realistic schedulers usually schedule globally or provide a mixture of global and local scheduling. Local scheduling by itself does not allow for fair load sharing among the processors in the system. Refer to Figure 1 for details concerning the flow of the scheduler.

Particular attention must be paid to how each process stream interacts with the others. How much processing should be allowed on one stream before work begins on the others? How do the separate address streams synchronize with each other given the states of the caches of the concurrently executing processors?

A logical separation of address stream execution allows one processor to work on an address stream from the beginning of the context switch to the end. This works in all cases where no data sharing occurs between separate processes and when processes run sequentially in the system. When a process exits a processor and is later rescheduled, the cache coherency transactions for its address space are guaranteed to be resolved for any memory reference occurring before the process starts. Thus every processor can run an address stream for a current process until it the stream switches to a new context. During execution of each context the address references are made consistent in the other caches such that when a process begins execution in another cache, its cache is guaranteed consistent.

One problem with this approach is that even unrelated processes share some amount of data in the kernel. Therefore, since coherency is not maintained on a cycle-by-cycle basis between all caches, some degree of error is introduced into the results. This problem is difficult to overcome without expanding the simulator to provide cycle-by-cycle simulation. This would run a great deal slower due to the added overhead of the more detailed model. In addition, a method is not available to produce kernel traces. As noted, in this study the targeted user programs contain no parallelism or data sharing. Thus, a process can only invalidate or update cache lines in other caches that were allocated on previous contexts of that process. In other words, process A cannot invalidate or update cache lines that were allocated to process B since they do not share data (address tags and pids will not match during snoopy look-up). The only way for process A to affect the cache lines of process B is to displace

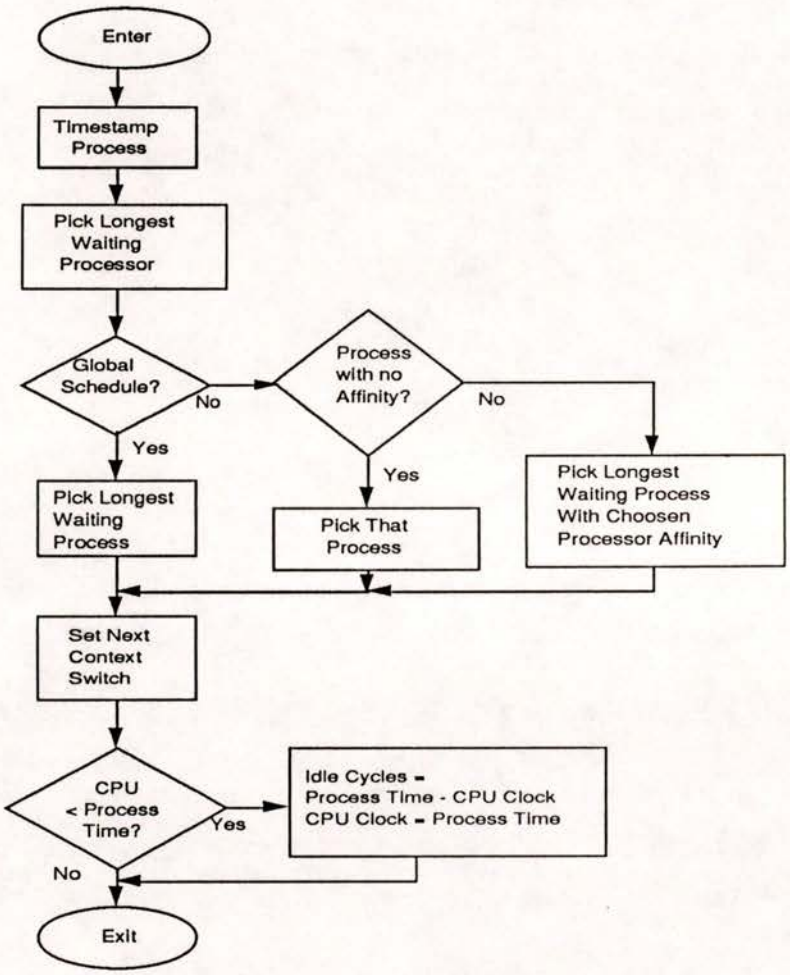


Figure 1: The Scheduler

them in the cache in which process A is currently executing.

Since a process is simulated in sequential order as dictated by the scheduling algorithm, we can guarantee that no execution of a process will precede (in the simulated context) a previous execution of that process. This is accomplished by time stamping a process when it is context switched out of a processor. Then if that process is scheduled to a processor whose clock (cycle count) is less than the time stamp, the processor's clock is set to the time stamp. This guarantees the sequential (with respect to cycle count) execution of the process. This argument when combined with the one from the paragraph above guarantees that misses due to invalidations can only occur when a previous instantiation of the process on another processor caused those invalidations.

Furthermore, we can detect when a miss was due to invalidation or displacement. This is accomplished by noting that even though simulation is not accomplished cycle by cycle, a processor's cache is consistent at the time a process is scheduled on it. Since a process cannot schedule itself on two processors at the same time there are no race conditions with respect to coherency. The question must be asked: Is there a race condition between invalidation of a line and another process' displacement of that line? This reduces to a don't care situation because upon completion of the time-slice the line will be displaced to the new value and pid whether the invalidation or displacement occurs first. The next process using the cache only knows that it missed due to displacement and not invalidation. For example, a cache line in processor 1 is valid from a previous instantiation of process A (tag = "pid + tag"). Now process B is executing on processor 1 and process A is executing on processor 2. If process A invalidates X and then process B allocates that line, the resulting state of the cache line is that of the allocation of process B. Conversely, if process B displaced X first then the invalidation from process A occurred, the invalidation would not match the new tag so the cache line would still contain the valid reference of process B. In either case the result is the same, a valid cache line for the reference of process B. For these reasons we can justify processes executing for their entire time-slice without cycle by cycle synchronization.

Coherency Details

This study examines snoopy based coherency algorithms since they are more flexible and are more prevalent in today's computers. The two basic types of snoopy protocols are invalidation and update protocols. We chose to simulate both and compare their effects on cache migration. The invalidation protocol is represented with a "Berkeley" type of coherency while the update protocol is represented with a "Firefly" type of protocol [8]. The Berkeley approach is a write-back protocol that maintains coherency on writes by invalidating matching lines in other caches. Digital Equipment Corporation's "Firefly" approach is also a write-back protocol but maintains coherency on writes by updating the matching cache lines in the other caches.

BENCHMARK PROGRAMS

Four benchmarks were chosen for simulation. Each of these was compiled and the object code annotated to generate address traces. The address traces of each program were fed into the simulator via a UNIX^{TM} pipe. The simulator then interleaved the traces corresponding to the type of scheduling selected. All programs ran in a UNIX^{TM} environment on a DECstation 3100^{TM}. Below is a description of each program; Table 2 provides characteristic data for each.

awk A UNIX^{TM} pattern scanning and processing language which scans a file for matching patterns and executes associated actions upon a match.

diff The UNIX^{TM} differential file comparator utility. This command compares the contents of two files and lists their differences.

dry2 The Drystone benchmark written in C. 50,000 iterations of this program were executed.

nroff A UNIX^{TM} text formatter. This command formats the text of a given file.

Benchmark	I-references	D-reads	D-writes
awk	32388725	4871777	3158560
diff	30939027	5870564	2582993
dry2	30254676	7060852	2858599
nroff	32036008	5983441	2701127
total	125618436	23786634	11301279

Table 2: Benchmark Programs

RESULTS

This section discusses the results of three experiments. In each experiment, simulations were conducted on a unified, direct-mapped cache with a line size of four words. Four pairs of scheduling and coherency combinations were modeled in each experiment: Global Berkeley, Local Berkeley, Global Firefly, and Local Firefly. As previously described, local scheduling refers to adding a processor affinity to each process and only allowing it to run on that processor. Global scheduling sends each process to the first available processor in a round-robin fashion. The cache coherency algorithms are modeled after the Berkeley and Firefly approaches described in a prior section. Each experiment was performed on models containing one to four processors. The measurements focused on three areas: total cache misses, transient cache misses, and misses due to invalidations. In addition, read and write miss measurements were recorded for the last two experiments.

The first experiment set the cache size to 32K words and applied the hashed virtual address traces of the four benchmarks to the simulator. Figures 2 through 4 illustrate these results. The second experiment increased the cache size to 128K words with everything else remaining the same. Figures 5 through 9 illustrate the cache performance resulting from the simulations of this larger cache. Finally, an experiment modeling a virtual 128K word cache was performed by applying the unhashed traces to the cache configuration of the second experiment. Its results are presented in Figures 10 through 14.

Cache Sizes

As expected, larger caches reduce the amount of cache misses across all scheduling and coherency algorithms. Larger reductions are seen with respect to the global scheduling and Firefly algorithms. As seen from Figure 5, the three processor 128K word cache used with the Global Firefly algorithm reduced cache misses by 46 percent over the same configuration with a 32K word cache (Figure 2). Large caches are effective in reducing miss rates in global schedulers since they greatly decrease the amount of transient misses. Recall that the causes of transient misses are interference by processes displacing each other's cache lines or initial cold start misses. Since larger caches tend to spread out the processes in the cache, less interference is seen. As the cache size grows, the number of transient misses due to global scheduling approaches the number of misses generated by local schedulers thereby alleviating some of the global scheduling penalties. Compare Figure 3 and Figure 8 to see how the number of transient misses due to global scheduling approach the number of transient misses due to local scheduling.

The Berkeley approach, although also benefiting from the reduction of transient misses, does not reduce its miss rate as drastically in larger caches. This is due to the additional cache misses from invalidations. In fact, by increasing the cache size, more misses are generated from invalidations from other processors. This is seen by comparing Figures 4 and 9. Larger caches are more likely to contain cache lines which match during an invalidation cycle thereby creating a miss if referenced later. The update approach generates no miss in this case since the cache line is kept active by updating the data.

Both the amounts of invalidates and updates increase with the larger caches during process migration. Invalidate traffic increases a modest amount due to the increase of shared-dirty states in the caches as processes migrate from cache to cache. The larger caches retain more data thereby keeping cache data around for longer periods of time. Update traffic increases by a slightly larger amount as the cache size increases. This is primarily due to the larger caches having a greater likelyhood of retaining data. Since the data has a greater chance of surviving in larger caches, more updates are required to keep the data coherent. Only when

no other cache contains a copy of the cache line may updating stop for that particular line.

Scheduling

Scheduling was shown to have the greatest impact on the total cache miss rate. It should be noted that when the processor count reaches four, even the global algorithms begin to schedule fairly locally. This is due to each process having a processor all to itself. When a process' context expires, no other process is waiting, so it gets scheduled back on the same processor immediately. This is true until one of the programs finishes execution, at which time two processors would be free upon each context switch. The global scheduler then schedules the switched out process to the "other" processor since it was waiting the longest (load sharing).

As seen from Figures 2 and 5, local scheduling provides the greatest reduction of total misses especially with larger numbers of processors (discounting the four processor case). In many cases, local scheduling reduces the number of misses by one-half or more compared to the corresponding global scheduler. As the number of processors increases, the amounts of transient misses and misses due to invalidations increase for global scheduling (discounting the one and four processor case). The global scheduler sends a process to more and more processors between successive visits to the same cache. This tends to increase the number of transient misses resulting from processes interfering with each other by displacing one another's data and by increasing the amount of cold start misses (see Figures 3 and 8). It also increases the amount of invalidations to each process' footprint since each process spends a smaller amount of time on each processor. This becomes more of a problem for small caches since the contention for the smaller number of cache lines increases. Local scheduling is beneficial in systems with large numbers of processors since it prevents this type of behavior. This is encouraging from a scheduler design perspective because there is the potential for a large reduction in the number of cache misses.

The Berkeley protocol benefits from local scheduling the most. Since the Berkeley protocol uses invalidations to keep the multiple caches

coherent, reducing the amount of invalidations will help increase the amount of data retained in each cache for each process. Local scheduling eliminates invalidation of cache lines (not invalidate traffic) by keeping the process on one processor as shown by Figures 4 and 9. A data line only gets invalidated if another process writes to that address. Since sharing was not examined in this study, no other process would generate writes to that address. Therefore, the Berkeley protocol benefits from local scheduling by a reduction of transient misses and by an elimination of misses due to invalidations.

Local scheduling greatly reduces the invalidate traffic, sometimes by as much as 90 percent over global scheduling. This difference grows as the numbers of processors increase. In fact, as the number of processors increase, the number of invalidates will increase for global scheduling while they decrease for local scheduling. This is explained by the amount of hits to dirty data lines in the caches. While a process runs out of a particular cache, writes to dirty data lines require no invalidations. Once a process migrates to a new cache, writes to what were once dirty cache lines now require invalidations to purge the other caches of matching lines. Local schedulers, on the other hand, do not generate this added invalidation traffic due to migrating dirty lines. They do generate extra traffic when dirty lines are displaced by other processes if future writes are made to the displaced address. Even so, local schedulers were also seen to reduce displacement of data over their global counterparts.

Updates are completely eliminated in a local scheduler since no process ever migrates to another cache. Global scheduling does generate update traffic which increases as the processor count increases. As processors are added, there is a greater likelihood that a migrating process has left an active line in one of the caches since the only way to purge lines from past caches is for other processes to displace them. An update is only required if another cache contains the data line the current cache is writing. Since increasing the number of processors increases the likelihood of a process' lines existing in other caches, the number of overall updates must increase.

Cache Coherency

The Firefly coherency protocol does the best with respect to reducing cache misses in the global scheduling approach. As seen from Figures 2 and 5, the Global Firefly approach reduces the amount of misses up to 54 percent over the Berkeley method. This is primarily due to Firefly generating no invalidations. Thus no cache misses are due to invalidates in the Firefly approach, while the Berkeley approach generates a sizable amount of misses due to invalidates. The amount of transient misses between the two approaches are practically identical as would be expected since transient misses are a function of the scheduling algorithm.

Local scheduling evened the differences between the miss rates of the Berkeley and Firefly protocols. Since the transient miss behavior is the same between the two (Figures 3 and 8), the only difference lies in misses due to invalidations. While Firefly never generates invalidations, local scheduling reduces misses due to invalidations in the Berkeley approach to zero. Therefore local scheduling causes both Berkeley and Firefly to generate the same amount of transient and invalidation misses resulting in the same overall miss behavior.

The miss rate of Firefly verses Berkeley is misleading since Firefly pays a large overhead to prevent invalidations. On every write to a word that is in multiple caches, an update must be performed. As a process migrates from processor to processor, the amount of duplicate cache lines increases between the multiple caches. Since multiple caches will retain each process' cache lines until they are displaced, updates will have to be performed while multiple copies exist. As the number of processors increases the probability that a cache line does not exist in multiple caches decreases. This gives rise to an abundance of update cycles to keep all the caches coherent. The Berkeley protocol, on the other hand, invalidates all the other duplicate cache lines after the first write of a process to the duplicate line. From then on, the cache line is dirty and resides only in the current processor. From a process migration point of view, this smaller number of invalidations verses updates gives the Berkeley protocol the advantage with respect to the number of bus transactions required to keep the caches coherent.

Local scheduling greatly reduces the amount of invalidations per-

formed in the Berkeley protocol as the number of processors increase. This is primarily due to fewer processes executing on each processor. This reduction produces fewer displacements in each cache. This will increase the amount of write hits to dirty data lines and thereby reduce the amount of invalidate traffic. Local scheduling eliminates all the update traffic generated by Firefly since there is no way for duplicate lines to exist in multiple caches. Therefore, in a local scheduler, the Firefly approach would generate fewer bus consistency cycles compared with the Berkeley approach.

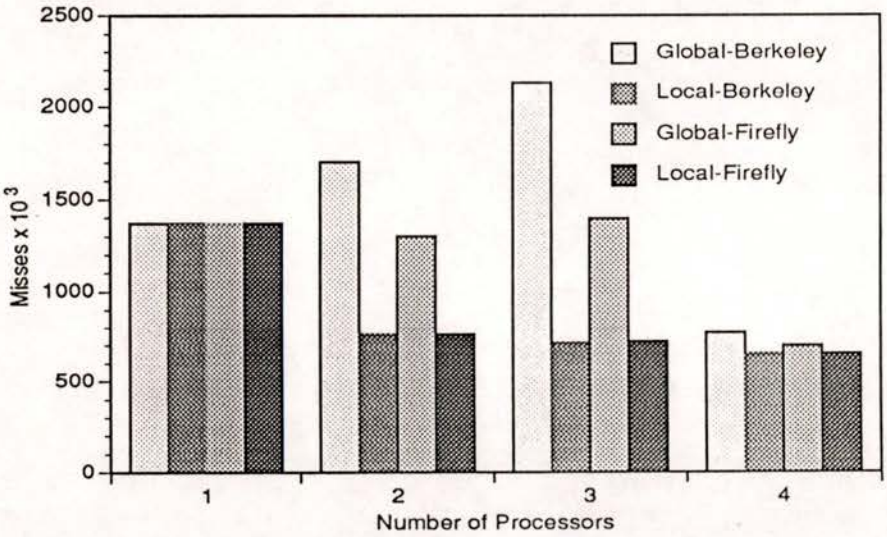


Figure 2: Total misses in a 32K physical cache.

Virtual Verses Physical Cacheing

A virtual cacheing strategy produced consistently more cache misses over the physical caches in every instance except for the four processor cases. This occurs because virtual addresses between processes conflict more than physical addresses. Physical addresses are spread out by the paging algorithm while virtual addresses tend to occupy the same page space between processes. This is especially true within the instruction space

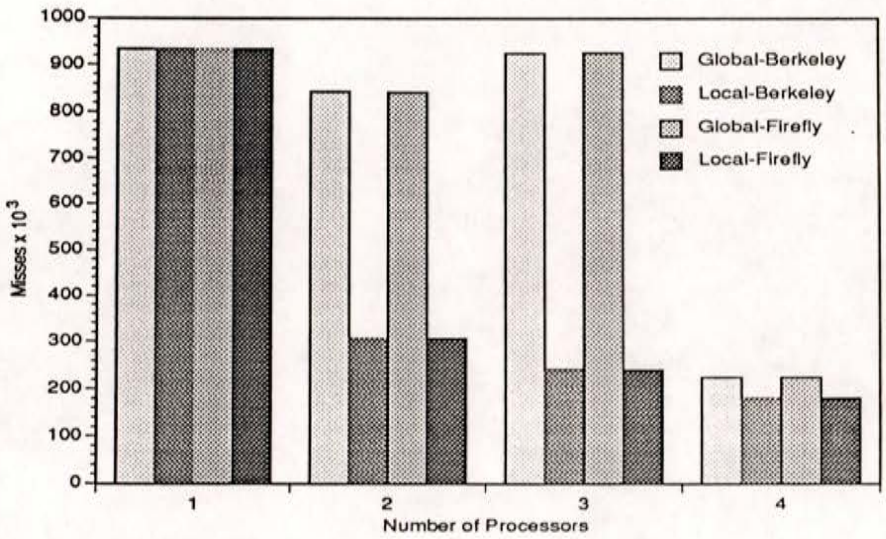


Figure 3: Transient misses in a 32K physical cache.

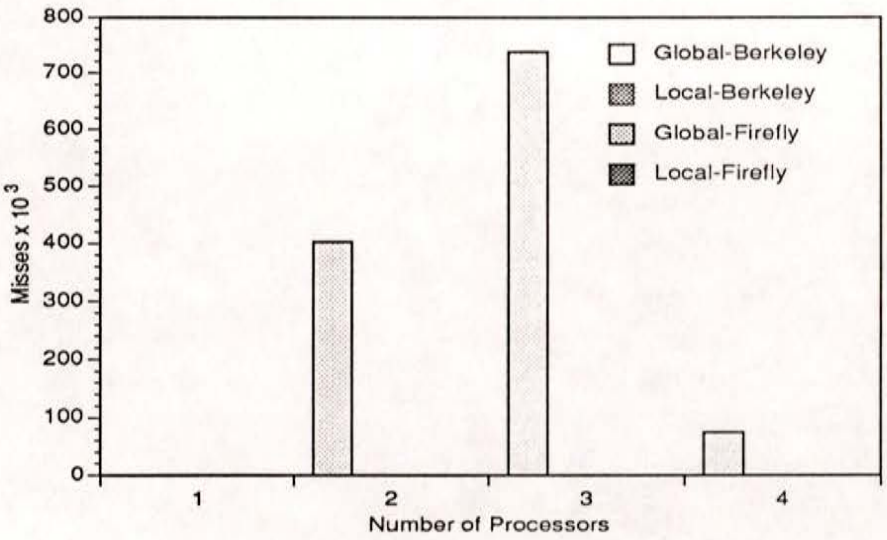


Figure 4: Invalidate misses in a 32K physical cache.

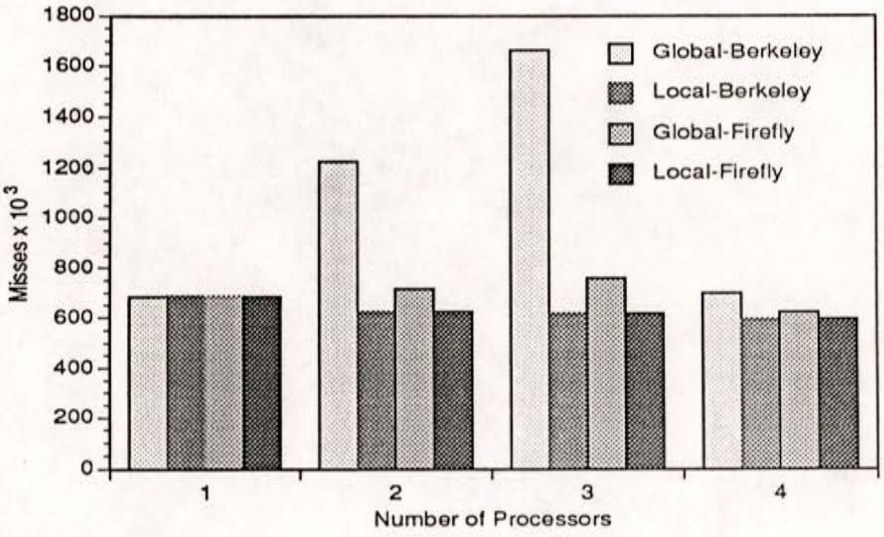


Figure 5: Total misses in a 128K physical cache.

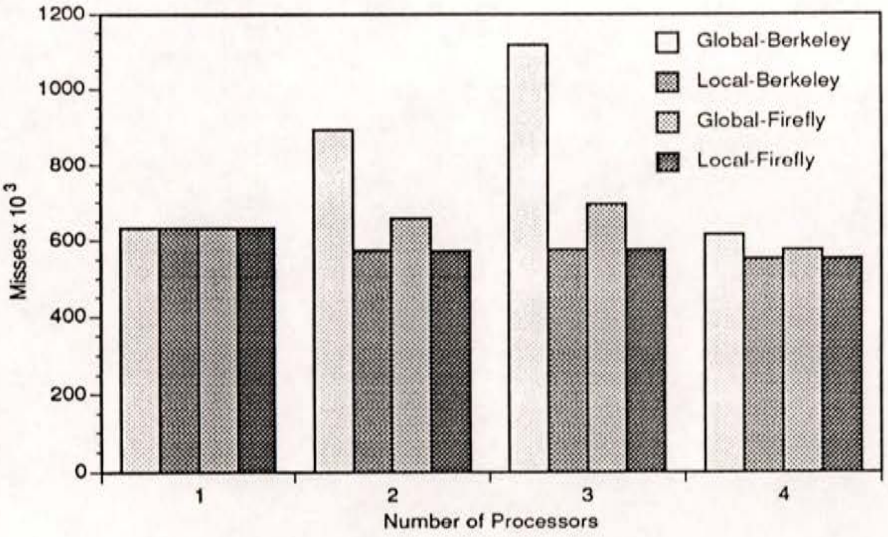


Figure 6: Read misses in a 128K physical cache.

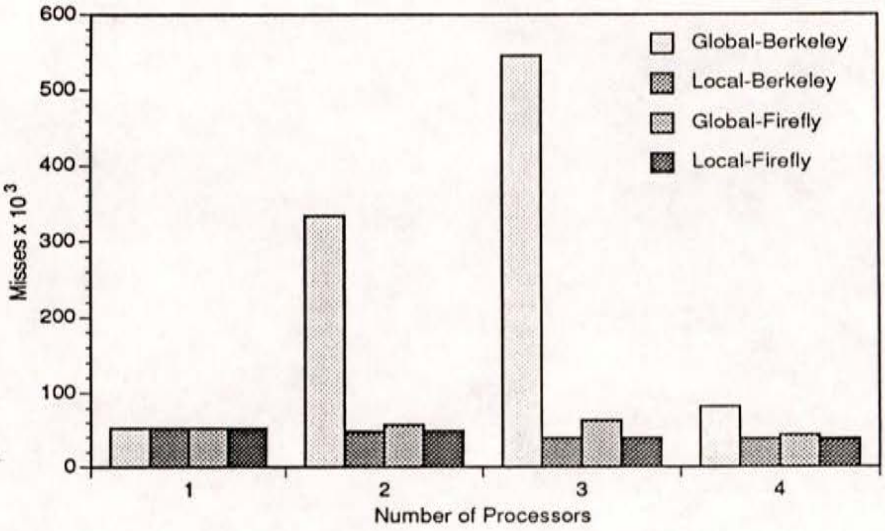


Figure 7: Write misses in a 128K physical cache.

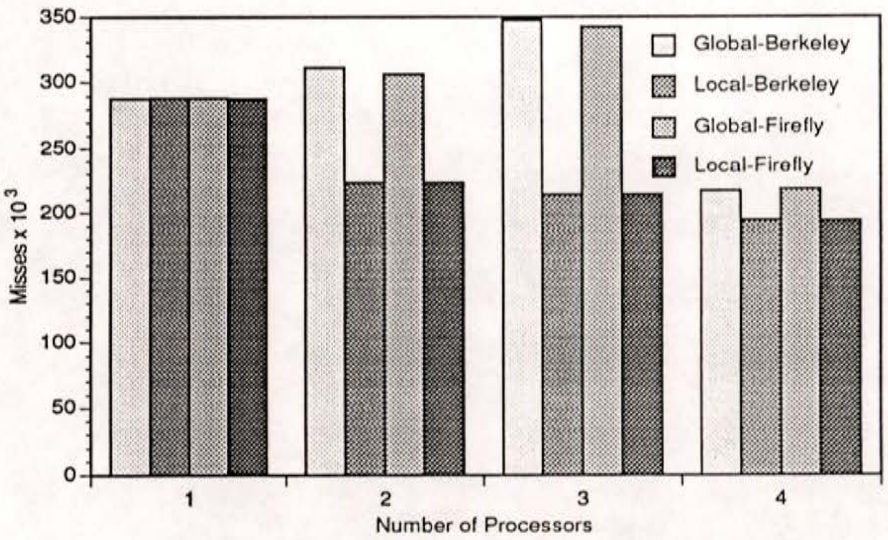


Figure 8: Transient misses in a 128K physical cache.

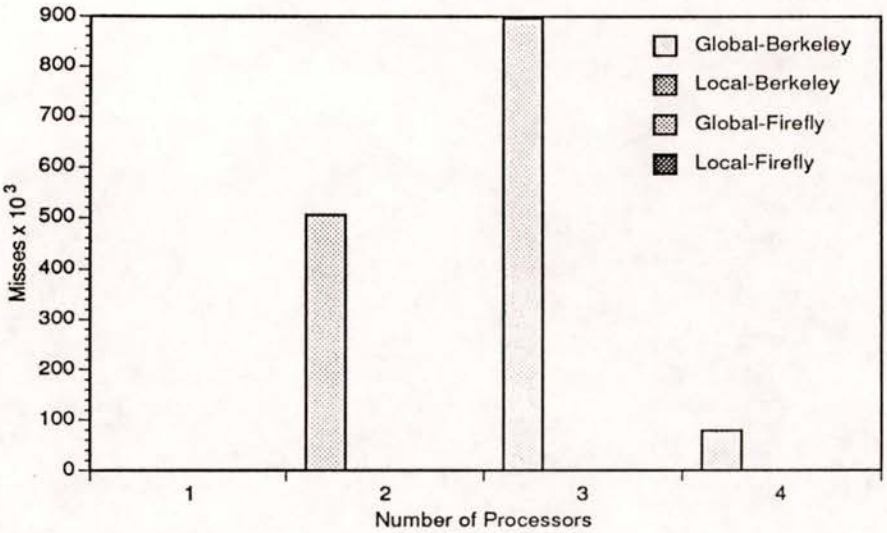


Figure 9: Invalidate misses in a 128K physical cache.

since most code is condensed at the beginning of each virtual address space. The read case in Figure 11 illustrates this well when compared to the write case in Figure 12 and the corresponding graphs in the physically addressed 128K word cache (Figures 6 and 7). The read case produced many more misses relative to the physically addressed cache while the write case produced a modest increase over its physical counterpart. This is primarily due to the interference of the virtual I-streams and the virtual stack regions of the processes. The virtual stack regions also start at the same location in virtual address space. The interference of virtual address spaces can clearly be seen by examining the transient misses of the virtual cache in Figure 13 and comparing them with the misses of the physical cache in Figure 8. The transient misses are much higher in the virtual cache with the exception of the four processor simulation.

One should expect that the global scheduling of a large number of processors would aggravate the difference between physical and virtual caches even more because virtual caches suffer from greater process in-

terference than physical caches. This is indeed true and can be seen by comparing the two graphs of total misses for virtual and physical caches, Figures 5 and 10. With three processors the global scheduler for the virtual cache produces twenty-five percent more misses than its physical counterpart while only five percent more with the local scheduler. Since Firefly generates no misses due to invalidates, its percentage difference is greater when compared with Berkeley (63 percent more misses in the virtual cache).

In the four processor case, each process had a processor to itself for most of its execution time. Only when one process terminated would migration start up again. Since very little displacement from competing processes occurred in the four processor case, the only significant displacement of cache lines was from the process itself. The virtual caches reduced the amount of self displacement by keeping the page addresses contiguous. Physical caches tend have a random distribution of page addresses and can cause greater amounts of displacements as pages are mapped so that they compete for the same cache line. While this is detrimental when there is little interference between processes, this miss rate penalty is more than offset when there is significant interference between competing processes as seen above.

Physical caches generate more update traffic than the corresponding virtual caches in multiprocessor environments. This can probably be attributed to the higher interference of virtual caches. Since it is less likely for data to remain in a cache (higher interference), it is less likely that an update is required.

CONCLUSIONS

This study examined process migration effects with respect to two scheduling algorithms and two cache coherency algorithms. Local and global scheduling were used to illustrate the extremes of scheduling algorithms and update (Firefly) and invalidate (Berkeley) based cache coherency schemes were chosen as the extremes of bus-based coherency.

The results illustrated that scheduling algorithms have a significant impact on the amount of cache misses in a multiprocessor system. As processes migrate they interfere with one another and increase the amount

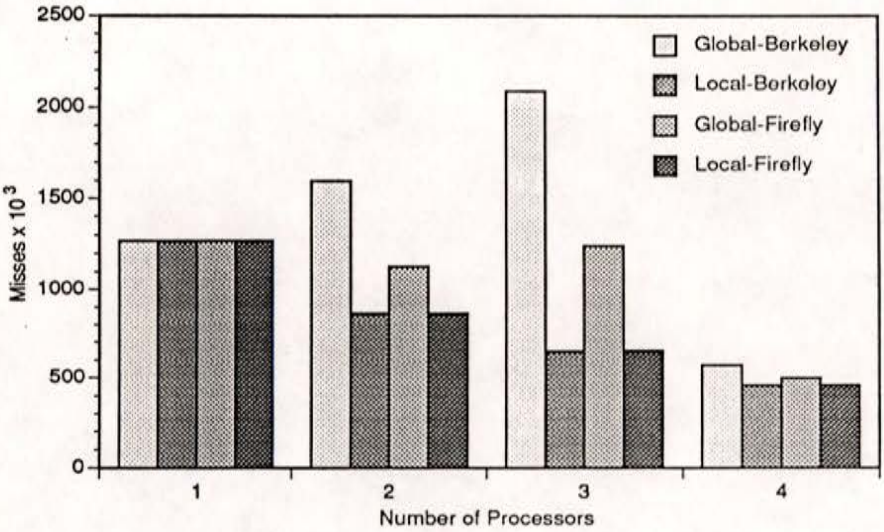


Figure 10: Total misses in a 128K virtual cache.

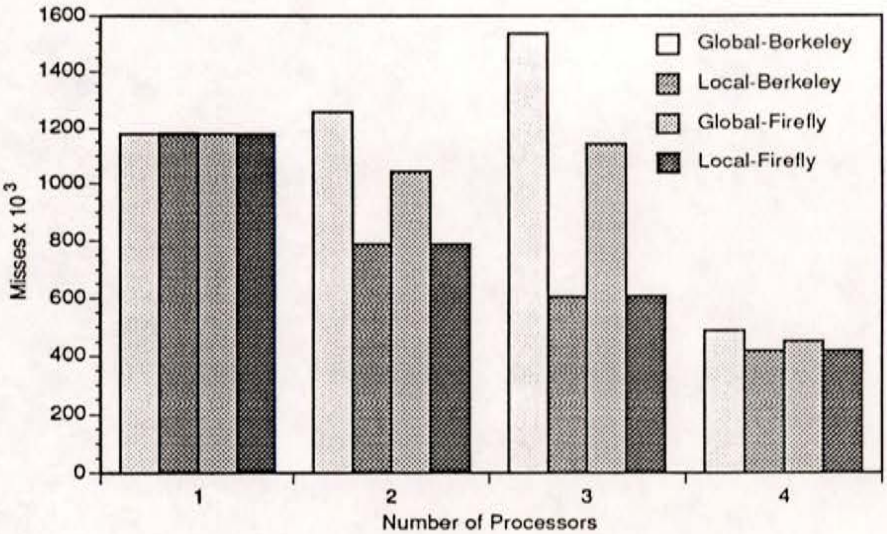


Figure 11: Read misses in 128K virtual cache.

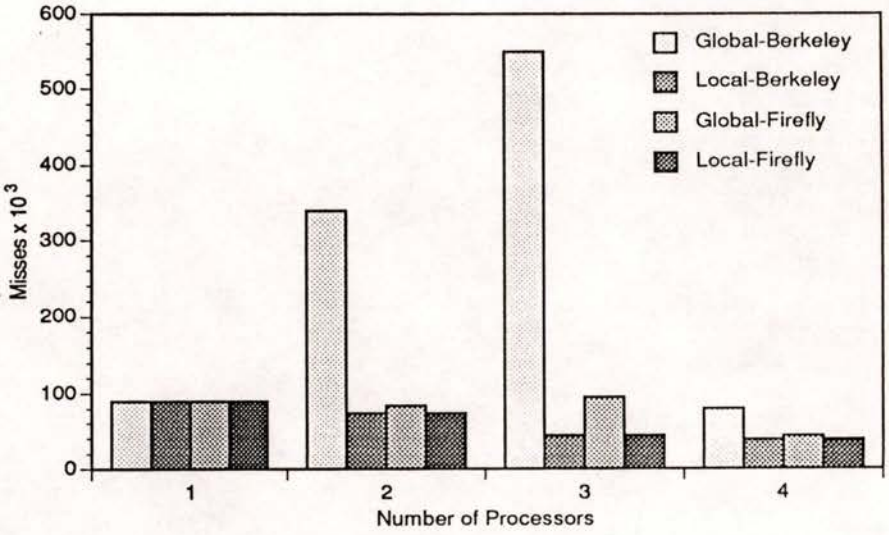


Figure 12: Write misses in 128K virtual cache.

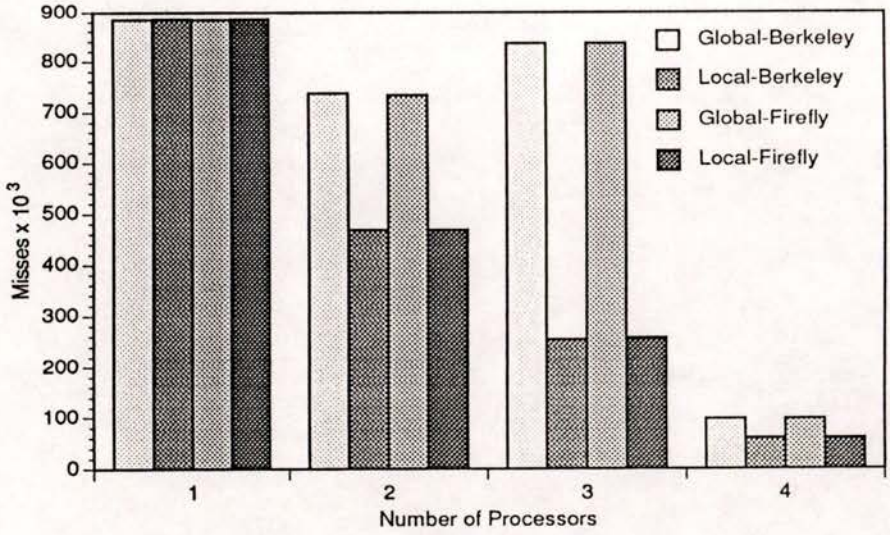


Figure 13: Transient misses in 128K virtual cache.

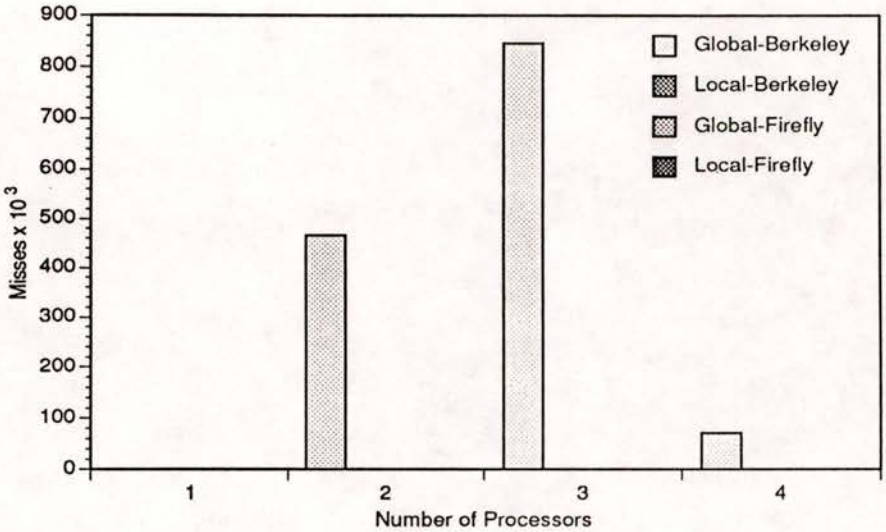


Figure 14: Invalidate misses in 128K virtual cache.

of transient misses per process. Local schedulers reduce these transient misses and eliminate the misses due to invalidations in write invalidate protocols. However, as cache sizes grow, the advantages of local scheduling are diminished because process interference is reduced as each process' addresses are spread across the larger caches. Since local scheduling eliminates misses due to invalidates, the advantages of update protocols over invalidate protocols are eliminated with respect to miss rates. On the other hand, local scheduling eliminates the update bus traffic while only reducing the invalidate traffic (the Berkeley algorithm does not determine when data is exclusively held by a cache).

Firefly provided the best miss rates for the global scheduling approaches since it generates no invalidations that could later force a miss. This is misleading though, since the elimination of invalidates requires update cycles to keep the cache lines consistent with each other. As seen from the global scheduling simulations, the amount of update cycles greatly exceeded the amount of invalidation cycles. With respect to local schedulers, both the Firefly and Berkeley protocols produced equal

miss rates. In this scenario, Firefly performed better since it generated no extra bus traffic to maintain bus coherency.

Virtual caches performed significantly worse when compared to the physical mapped (hashed) caches. Increased process interference was responsible for this and can be explained by the overlapping instruction and stack regions in the virtual address space. Physical mappings tend to randomly distribute page addresses in the caches which reduces process interference. Relative to physically addressed caches, virtual caches increase processor interference yet show no increase in misses due to invalidates. This tends to decreases the difference between update and invalidate coherency protocols because misses due to invalidations were responsible for this difference in the first place. Increased processor interference also makes local scheduling strategies even more attractive for virtual caches by providing processor affinity to each process.

FUTURE WORK

No tools were available to generate kernel traces in this study. Since many kernel routines execute during each benchmark execution, the address references made by these routines are likely to remain in each processor's cache, especially if the kernel address space is shared. A study of the multiprocessing behavior of programs with their corresponding kernel calls could provide a much greater understanding of the complete process migration effects on cacheing. Lower cache miss rates should be seen because more data in each cache is likely to be reused by each process.

Also, context switching was approximated using a Poisson distribution. In reality, context switches are produced by a combination of system calls and time-slice expirations. A more accurate model of context switch behavior could identify which system calls generate a context switch. This model could isolate which addresses in the trace correspond to these system calls and generate a context switch when such an address is encountered. The model would still implement a context switch due to time-slice expiration for the case when no change of context occurs before the time-slice expires.

Another path of investigation corresponds to Stone's [1] assertion that

cache misses can be reduced if the processor stops allocating cache lines when a process is close to the end of its time-slice. This type of behavior would be fairly easy to investigate with the tools developed for this study.

Finally, the simulator used here could easily be modified to a particular instance of a scheduling algorithm. Operating system designers then can create their favorite scheduler and simulate it with these tools. The benefits or limitations of such an new algorithm can be compared with the more conventional algorithms.

ACKNOWLEDGEMENTS

The authors wish to thank Maureen Ladd for her help in preparing this chapter and the Digital Equipment Corporation, who supported Andrew Ladd on a GEEP scholarship.

References

[1] H. Stone. Cache allocation strategies for competing processes, February 1990. Presentation in Distinguished Lecture Series at University of Michigan.

[2] S. Thakkar and M. Sweiger. Performance of an OLTP application on the Symmetry multiprocessor system. In *Proceedings of the 17th International Symposium on Computer Architecture*, volume 1, pages 228–238. IEEE, 1990.

[3] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3), Sepetember 1982.

[4] M. Kobayashi and M. MacDougall. The stack growth function: Cache line reference models. *IEEE Transactions on Computers*, 38(6), June 1989.

[5] A. Borg, R. Kessler, G. Lazana, and D. Wall. Long address traces from RISC machines: Generation and analysis. WRL research report 89/14, Digital Equipment Western Research Laboritory, Palo Alto, California, September 1989.

[6] M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, volume 1, pages 348–354. IEEE, 1984.

[7] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proceedings of the 11th International Symposium on Computer Architecture*, volume 1, pages 355–362. IEEE, 1984.

[8] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[9] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, volume 1, pages 280–289. IEEE, 1988.

[10] R. Clapp, T. Mudge, and J. Smith. Performance of parallel loops using alternative cache consistency protocols on a non-bus multiprocessor. In M. Dubois and S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pages 131–152. Kluwer, 1990.

[11] J. Larus. Abstract execution: A technique for efficiently tracing programs. Technical report, University of Wisconsin-Madison, February 1990.

[12] MIPS. *System Programmer's Package (SPP) Reference.* MIPS Corp., Boston, 1988.

[13] D. Samples. Mache: No-loss trace compaction. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, volume 1, pages 89–97. IEEE, 1989.

[14] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11), November 1988.

[15] R. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, volume 1, pages 186–195. IEEE, 1988.

JULY 1989

WRL Research Report 89/10

Integration and Packaging Plateaus of Processor Performance

Norman P. Jouppi

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	WRL-Techreports@decwrl.dec.com
CSnet:	WRL-Techreports@decwrl.dec.com
UUCP:	decwrl!wrl-techreports

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word "`help`" in the Subject line; you will receive detailed instructions.

Integration and Packaging Plateaus of Processor Performance

Norman P. Jouppi

July, 1989

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

Integration and packaging set limits to processor performance. This paper investigates the nature of these limits and their implications for computer architecture, organization, and design techniques.

This is a preprint of a paper that will be presented at the
International Conference of Computer Design,
IEEE, Cambridge, Massachusetts, October 2-4, 1989.

Copyright © 1989 IEEE

Table of Contents

1. Introduction 1
2. Is 3-Dimensional Packaging Needed? 1
3. The Relative Importance of Signal Integration 3
4. The Advent of Fully-Integrated Processors 6
5. The New Technology Paradox 7
6. Conclusions 7
7. Acknowledgements 7
References 8

List of Figures

Figure 1: Machine structure vs. packaging required 2
Figure 2: Plateaus of processor performance 6

List of Tables

Table 1: Internal and external nets vs. integration 3
Table 2: Model parameters for each level 4
Table 3: Different pipelining possible in LSI 5

1. Introduction

Many papers have been written about limits to computer performance. Some of these have been based on quantum-mechanical [8], thermal [8], or size [10] limits. More recently circuit and system performance limits on VLSI interconnections and packaging have been investigated [1]. This work refines integration and packaging performance limits specifically in the context of computer systems. In particular, limits of computer performance under various packaging, architectural, organizational, and design techniques (e.g., gate-array vs. custom) are explored.

2. Is 3-Dimensional Packaging Needed?

There are four components to delay when driving a signal off a package. First, there is a fixed driver delay dependent on the chip's technology for going off-chip. Second, once off-chip, there is also a delay proportional to the signal length traveled. This transmission delay is dependent on the voltage swings, transmission-line environment, and the actual distance traveled which is a function of the packaging technology. (These two delays are also present on-chip, but with smaller overall delay.) Third, there is a fixed receiver delay. Fourth, there is a delay due to clock skew between the transmitter and receiver. This can be a significant percentage of the total interconnection delay in high performance systems. In the remainder of the paper, interconnect delay will be used to describe the sum of these four components.

Since the interconnection delay has a term proportional to the distance traveled, the observation has been made that the fastest computers would be limited by their size. One obvious limit is that the radius of the machine should be smaller than the distance traveled in a clock cycle by a signal at the speed of light. More specifically, if some portion of a machine must communicate with other parts of the machine in one cycle, the size of the machine is limited by the speed of transmission in the packaging media used in the machine. This has led to further observations that the maximum volume contained within a radius is a sphere, and that the fastest machines must be spheres. This puts a high premium on the development of 3-dimensional or "volumetric" packaging techniques.

The drive for volumetric density has significant implications for computer packaging. Tall heat sinks waste machine volume compared to the volume requirements of only the chips and the boards. Therefore to maximize packing efficiency, parts must be stacked on top of each other leaving no room for heat sinks. Thus the search for volumetric packaging techniques leads to

liquid-based immersion cooling techniques [4], where components can be stacked together very closely.

In reality however, the need to make machines in the shape of spheres is only true if the machine consists of uniformly distributed interconnections between random points in the machine. This is not completely true for any machine, and is mostly untrue for most machines.

For example, consider a machine in the shape of a sphere. Imagine the machine is pipelined, and has independent functional units for floating-point operations. In this case machine performance would not be degraded if we "pulled" the floating-point pipeline out of the sphere (see Figure 1). This is because the data in a floating-point functional unit proceeds from pipestage to pipestage without communicating with the rest of the machine except at its input and its output.* Thus the machine is not limited to a sphere when the interconnections are structured and local instead of random.

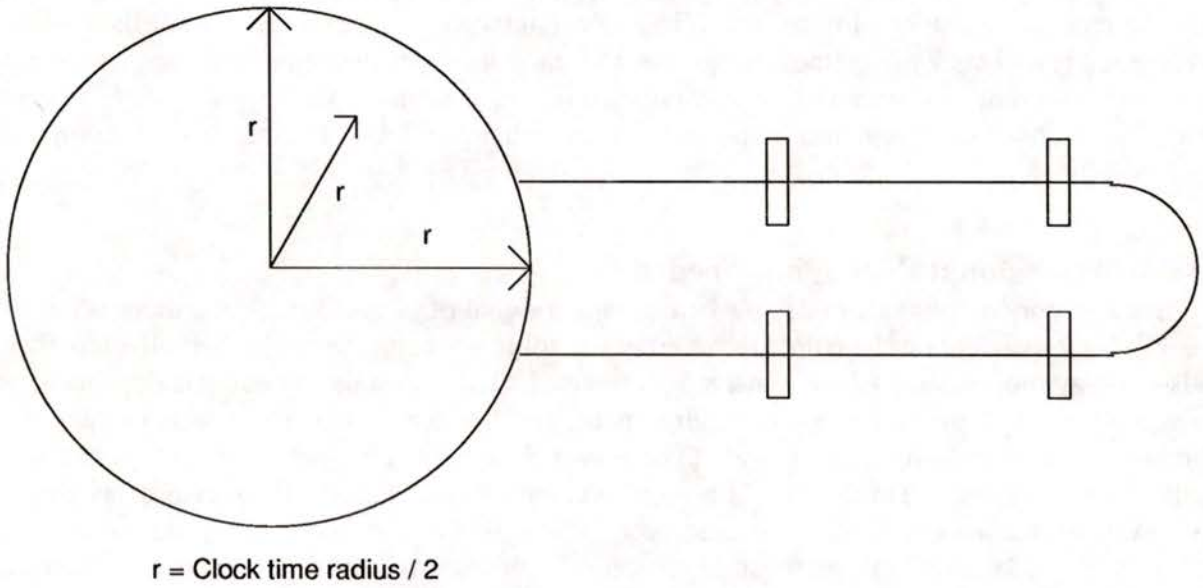


Figure 1: Machine structure vs. packaging required

Another important example of machine structure is provided by instruction and data caches. For caches of reasonable size, most instruction and data references are satisfied by the caches without recourse to main memory. Since cache misses are the infrequent case, access beyond the caches could be made to operate at a slower cycle time than the core of the machine without performance degradation. (Of course the overall access time and latency of the cache miss should still be minimized.) Similarly, if the CPU is entirely contained on a single chip (not including caches, floating-point, or MMU), many signal paths are entirely contained on the CPU chip. Some signal distribution frequencies based on the MultiTitan CPU chip [6] are given in

*We can clock the latches outside of the clock-time sphere in several ways. If the clock can be delayed with good control of skew, we could add a pipestage of delay to the clocks of the latches one clock time away, etc. Another method would be to use clock forwarding techniques, where the clock travels along with the data.

Table 1. This shows the very strong tendency towards interconnection "locality of reference." In fact, this interconnection locality of reference is a corollary of Rent's rule [9]. This is because as the number of gates on a chip increases, the percentage of nets that cross the chip boundary gets smaller and smaller. (This is also true for blocks within a chip.)

Unit	Total nets	External nets	Percent external
1 bit adder	41	13	31.7%
32-bit adder	1,120	106	9.5%
CPU datapath	11,263	148	1.3%
CPU + L1 ICache	62,745	136	0.2%
Above + FPU + MMU	160,000+	80	<0.05%

Table 1: Internal and external nets vs. integration

Thus, based on interconnection locality of reference and the structure inherent in computers, only a small portion of the machine is limited by speed of light considerations. Since signals travel at least 3in per nanosecond on most types of PC boards, this gives us a fair amount of room to work with. In particular, 3-D volumetric packaging techniques are unnecessary if the core of the machine fits within a circular instead of spherical clock-time radius, or if parts of the machine can extend beyond this radius.

Unfortunately this is not the whole story. If only 1% of the signals are not integrated into the core processor but they are on the critical path of the machine, the system performance will still be low. In particular, machines which contain no cache, such as Cray machines [2, 4], have problems in this respect. By not taking advantage of the locality of signal travel gained from the locality of data reference in a cache, main memory must be included in the core of the machine, greatly increasing the size of the core machine and the importance of advanced packaging techniques.

3. The Relative Importance of Signal Integration

At each level of integration the best performance is obtained by integrating critical paths of a machine onto the same chip. In addition, implementation of wide busses between functional units is simplified if both functional units reside on the same chip. Even if the best choices are made for the co-integration of circuits at a given density of integration, the delays in packaging will create limits to machine performance. Increased pipelining can partially compensate for packaging delays. For example, a pipestage could be added for each chip crossing, such that in the limit, the cycle time is equal to the maximum interconnection delay between chips. (In fact, the longest wires could be further pipelined by inserting additional latch chips, giving a cycle time closer to the average wire length.) But deeper pipelining also results in more cycles lost due to breaks in the pipeline as a result of branches and data dependencies. In the limit, the performance of the machine will be determined by the number of chip crossings required by the data and control (i.e., branch) dependencies present in a program.

To get a better understanding of these limits, we simulated machines with with infinitely fast gate delays while varying the density of integration. This provided an upper bound on processor performance. Eight programs were simulated with a parameterizable machine architecture and compilation system [7]. This system optimizes, reorders, and simulates program code based on

the pipeline characteristics specified in a configuration file. The programs simulated were our C compiler, PC board router, Linpack, the Livermore Loops, a timing verifier, the Hennessy Stanford benchmarks, whetstone, and the Unix Yacc program.

The latencies of functional units in a series of machines were estimated based on the number of chip crossings likely for each level of integration. Table 2 lists some assumptions for typical machines at each level of integration. For example, in an LSI environment a 32-bit adder might be contained on a single chip. But to use the results of an addition in another computation, we might have to leave the adder chip and go through a chip containing result forwarding (i.e., bypass) logic. Thus the total add delay in this technology involves crossing from the adder to the bypass back to the adder, for a total of two interconnection delays. In a VLSI single-chip CPU, the integer adder and bypass should be on the same chip. Since we are assuming gate delays are zero, the integer add delay on a VLSI chip is zero. However, the floating point unit might contain separate chips for addition/subtraction, multiplication, division, and floating-point registers and bypassing. Then two chip crossings would be required per floating-point operation. The only other chip crossings for a VLSI CPU without on-chip caches would be those to fetch instructions or to fetch data during load instructions. Finally, consider an ULSI machine which has its first-level instruction and data caches on-chip along with the MMU and floating-point support. It would only incur off-chip interconnection delays for cache misses. Two ULSI machines were considered, one with small on-chip caches and one with large on-chip caches.

Characteristic	LSI	VLSI	ULSI#1	ULSI#2
Chip crossings per:				
Integer add	2	0	0	0
Load + addr gen	4	2	0	0
FP ops	8	2	0	0
branch on CPU reg	4	0	0	0
1st level cache:				
On-chip or off	off	off	on	on
Instruction	4KB	64KB	1KB	8KB
Data	8KB	64KB	2KB	16KB
2nd level cache:				
Mixed	none	none	4MB	4MB
Crossings on miss			20	20

Table 2: Model parameters for each level

At each level of integration, a family of machines with different pipeline depths were simulated. For example, consider the pipelines possible with an LSI technology (see Table 3). The machine cycle time was varied between different multiples of the chip-to-chip interconnection delay. For example, a machine with a cycle time equal to a single chip crossing must have latches on each chip input pin. A machine pipelined this heavily will have many pipeline stages in technologies with low levels of integration. Similarly, a machine with a cycle time four times that of the interconnection delay has a relatively small number of pipeline stages even at lower levels of integration.

Figure 2 shows the results of simulations based on this model. The curves drawn are the average performance over all benchmarks. Figure 2 shows that for each level of integration, the interconnect delay sets a limit to machine performance. This is because although the cycle time

	Cycle time in interconnection delays			
Characteristic	4	2	1	0.5
Cycle latency per:				
Integer add	1	1	2	4
Load + addr gen	1	2	4	8
FP ops	2	4	8	16
branch on CPU reg	1	2	4	8
1st level cache:				
Cycles on miss	5	10	20	40

Table 3: Different pipelining possible in LSI

may be kept small, the extra pipestages that were added hurt performance because of data dependencies and their resulting breaks in the pipeline. Each curve for a given level of technology saturates when most of the performance is already being lost in pipeline breaks. This results in little or no additional performance when the number of pipestages is increased by decreasing the cycle time (i.e., for points on the curve farther to the right.) For a given cycle time (e.g., equal to one interconnect delay), the effects of increasing levels of integration can be seen by moving from a lower-integrated curve to a higher-integrated curve. Finally, if one interconnect delay (i.e., one on the lower axis) is taken to be 10ns, the relative performance figures correspond roughly to processor MIPS.

Packaging also affects the limit of machine performance. However, most packaging technologies are only a second-order effect compared to the level of integration. This is because better packaging can reduce the component of interconnection delay due to wire length, but the packaging technology generally does not affect the fixed delay component due to the off-chip driver. (Certain wafer-scale integration technologies can reduce this in the limit, however.) Thus variation in packaging technology only allows system performance to vary from one plateau of performance to about the next or previous plateau, depending on whether it is better or worse than average packaging.

Note that the x-axis of Figure 2 is specified in units of interconnection delay. Thus systems with high performance signal transmission (e.g., unidirectional single-driver single-receiver terminated transmission lines) will have significantly higher performance plateaus than systems with poor signal transmission (e.g., multi-driver multi-receiver unterminated TTL traces).

Finally, there are a number of other factors which affect the importance of packaging in system design besides integration and whether the machine is cache-based. Some second-order effects that can affect the amount of circuitry integrated on a chip by about a factor of two each are whether the machine is a CISC or a RISC machine, whether the design is gate array or full custom, and whether the implementation is complex or is simple and straightforward. Thus advanced packaging would be least important for a straightforward full-custom implementation of a RISC, due to its economical use of chip real estate that allows maximum integration of functional units at a given level of technology. Advanced packaging will be more important for complex gate-array implementations of CISC machines.

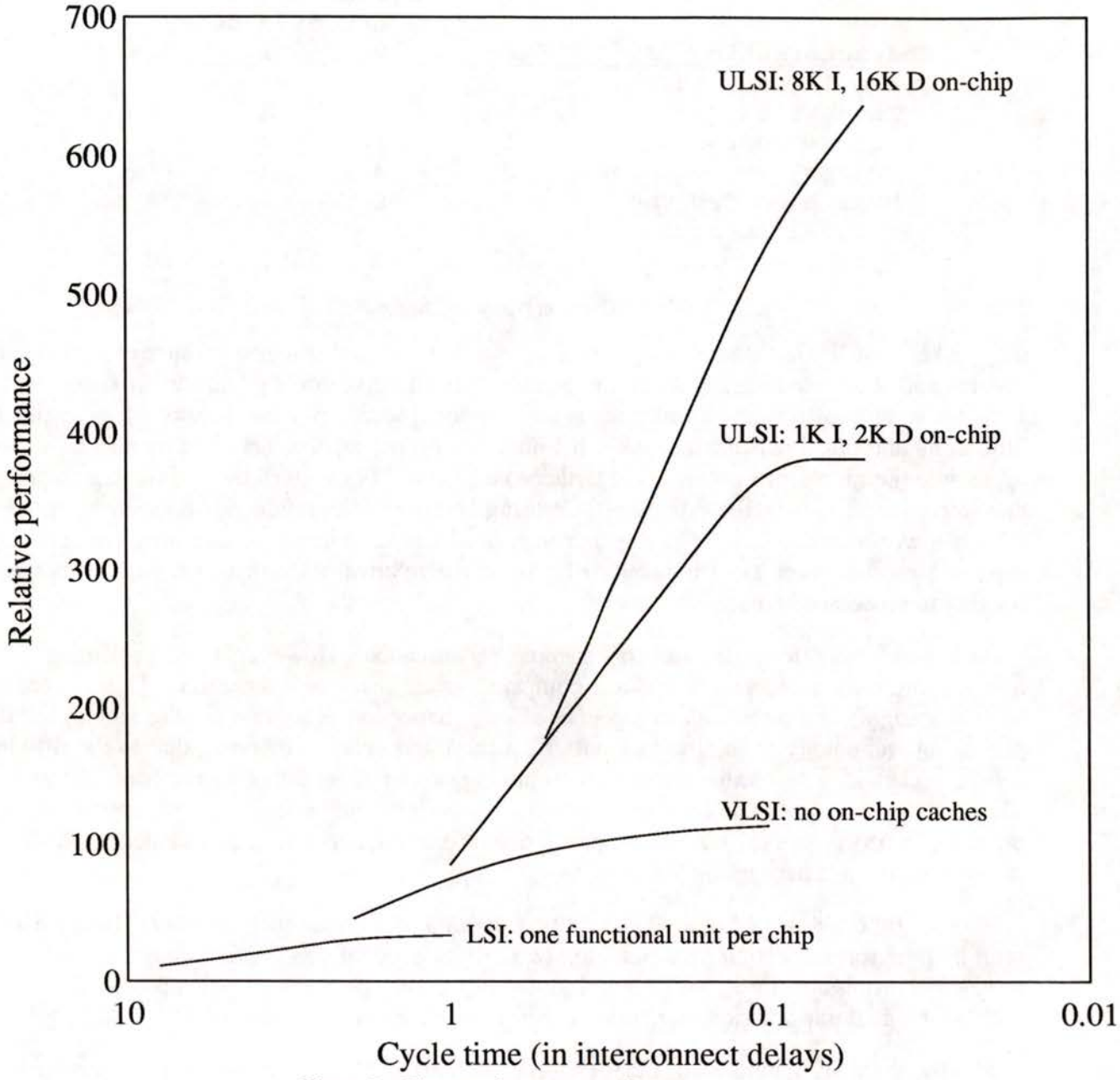


Figure 2: Plateaus of processor performance

4. The Advent of Fully-Integrated Processors

The most interesting plateaus in Figure 2 are the top two. At these levels a CPU, FPU, MMU, instruction and data cache have all been integrated onto a single chip. These two plateaus are significantly higher than the others because at this level of integration most instructions can execute entirely on chip. Thus for most instructions, they are *fully-integrated processors*.

Fully-integrated processors have many benefits. First, since only cache miss and write-back traffic crosses the pins, for reasonable on-chip cache sizes the pins can operate at a significantly lower frequency than the on-chip clock rate. For example, the off-chip cache refill could occur

with a 10ns clock cycle, while the on-chip clock period could be 2.5ns. If the 2.5ns internal clock is produced by frequency synthesis from the external 10ns clock via a phase-locked loop, no signals with fast edge rates or cycle times under 10ns need appear at the chip's pins. This allows rather conventional packaging to be used compared to that required to support a machine built from many chips operating with 2.5ns chip crossings.

5. The New Technology Paradox

New technologies with very small gate delays are initially available only with relatively low levels of integration. As more and more circuits of a new technology are built, technology travels down an integration learning curve of increasing density. For example, the first available GaAs chips had significantly fewer gates per chip than CMOS or ECL chips in the same time frame.

Thus in order to exploit emerging technologies, advanced packaging technology must be used to partially ameliorate the effects of the lower density available in the emerging technology. For example, the Cray-3 needs to attain very dense packaging because its GaAs chips only contain 200-400 gates of logic [5]. Depending on the chip technology advantages and the packaging technology available, the overall system performance of a machine built with an emerging technology at a low level of integration could still be less than that obtainable with a highly or fully-integrated version of the current mature technology. This discourages the widespread use of emerging technologies, and slows their progress along technology learning curves. An example of this is Josephson junction technology. Although IBM was able to built a prototype Josephson junction-based system in the late 1970's with average gate delays of 44ps [3], the fastest overall cycle time for a system of four chips on two cards was 3.7ns [11]. This is a machine cycle time approaching 100 gate delays, which is far more than the 8 and 4 gate delays per cycle obtained in the Cray-1 and Cray-2 [4].

6. Conclusions

Integration and packaging set limits to processor performance. If cached-based architectures are used, only a relatively small core of a machine is limited by interconnection transmission delays. If large levels of integration are used to build cache-based machines, the need for volumetrically dense packaging is practically eliminated. Machines that contain a CPU, FPU, MMU, and instruction and data caches all on one die are fully-integrated processors from the standpoint of most instructions. Fully-integrated processors can have modest electrical signal I/O requirements because the frequency of signals crossing their pins can be several times less than that of the on-chip clock frequency. Finally, in order to exploit emerging technologies without high levels of integration, advanced volumetric packaging techniques will still be very important.

7. Acknowledgements

The author wishes to thank Richard Swan and Bill Hamburgen for stimulating discussions on these issues. David Wall provided the paramterizable simulation system used in this study. Many members of the Western Research Lab and Chuck Thacker of the Systems Research Center provided valuable comments on an early draft of this paper.

References

[1] Bakoglu, Halil B.
Circuit and System Performance Limits on ULSI: Interconnections and Packaging.
PhD thesis, Stanford University, 1986.

[2] Cray Research Inc.
The CRAY-1 Computing System Reference Manual.
Chippewa Falls, WI, 1976.

[3] Gheewala, T.
Josephson-Logic Devices and Circuits.
IEEE Transactions on Electron Devices ED-27(10), Oct, 1980.

[4] Hockney, R. W., and Jesshope, C. R.
The Cray X-MP and Cray-2.
Parallel Computers 2.
Adam Hilger, 1988, pages 146-155.

[5] Iverson, Wesley R.
It Looks Like an '89 Debut for the Cray-3 After All.
Electronics , Dec, 1988.

[6] Jouppi, Norman P., Tang, Jeff, and Dion, Jeremy.
A 20 MIPS Sustained 32b processor with 64b Data Bus.
In *The International Solid State Circuits Conference*, pages 7.5. IEEE, February, 1989.

[7] Jouppi, Norman P., Wall, David W.
Instruction-Level Parallelism for Superscalar and Superpipelined Machines.
In *The Third Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 272-282. IEEE, April, 1989.

[8] Keyes, Robert W.
Physical Limits in Digital Electronics.
Proceedings of the IEEE 63(5):740-797, May, 1975.

[9] Landman, B. S., and Russo, R. L.
On a pin versus block relationship for partitions of logic graphs.
IEEE Transactions on Computers C-20(12):1469-1479, Dec, 1971.

[10] Wilkes, Maurice V.
Keynote Address - Size, Power, and Speed.
In *The 10th Annual Symposium on Computer Architecture*, pages 2-4. IEEE Computer Society Press, May, 1983.

[11] Zappe, Hans H.
Josephson Computer Technology - An Overview.
In *The International Conference on Computer Design*, pages 516-517. IEEE, October, 1983.

WRL Research Reports

"Titan System Manual."
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

"Global Register Allocation at Link Time."
David W. Wall.
WRL Research Report 86/3, October 1986.

"Optimal Finned Heat Sinks."
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

"The Mahler Experience: Using an Intermediate Language as the Machine Description."
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

"The Packet Filter: An Efficient Mechanism for User-level Network Code."
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

"Fragmentation Considered Harmful."
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

"Cache Coherence in Distributed Systems."
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

"Register Windows vs. Register Allocation."
David W. Wall.
WRL Research Report 87/5, December 1987.

"Editing Graphical Objects Using Procedural Representations."
Paul J. Asente.
WRL Research Report 87/6, November 1987.

"The USENET Cookbook: an Experiment in Electronic Publication."
Brian K. Reid.
WRL Research Report 87/7, December 1987.

"MultiTitan: Four Architecture Papers."
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

"Fast Printed Circuit Board Routing."
Jeremy Dion.
WRL Research Report 88/1, March 1988.

"Compacting Garbage Collection with Ambiguous Roots."
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

"The Experimental Literature of The Internet: An Annotated Bibliography."
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

"Measured Capacity of an Ethernet: Myths and Reality."
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

"Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description."
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

"SCHEME->C A Portable Scheme-to-C Compiler."
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

"Optimal Group Distribution in Carry-Skip Adders."
Silvio Turrini.
WRL Research Report 89/2, February 1989.

"Precise Robotic Paste Dot Dispensing."
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

"Simple and Flexible Datagram Access Controls for Unix-based Gateways."
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

"Spritely NFS: Implementation and Performance of Cache-Consistency Protocols."
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

"Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines."
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

"A Unified Vector/Scalar Floating-Point Architecture."
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

"Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU."
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

"Integration and Packaging Plateaus of Processor Performance."
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

"A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance."
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

"Leaf: A Netlist to Layout Converter for ECL Gates."
Robert L. Alverson and Norman P. Jouppi.
WRL Research Report 89/12, July 1989.

WRL Technical Notes

"TCP/IP PrintServer: Print Server Protocol."
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

"TCP/IP PrintServer: Server Architecture and Implementation."
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

SEPTEMBER 1989

WRL Research Report 89/14

Long Address Traces from RISC Machines: Generation and Analysis

Anita Borg
R. E. Kessler
Georgia Lazana
David W. Wall

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word "`help`" in the Subject line; you will receive detailed instructions.

Long Address Traces from RISC Machines:

Generation and Analysis

Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall

September, 1989

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

The accurate analysis of cache designs is becoming ever more important as processor speeds rapidly outstrip memory speeds and cache misses become a more significant factor in system performance. It can easily be shown that a factor of 10 decrease in processor cycle time with no change in the memory system may increase execution speed by only a factor of 2, the difference being due only to the increased relative cost of servicing cache misses.

The primary tool for cache analysis is simulation based on address traces of running systems. The accuracy of the results depends both on the simulation model and the accuracy of the trace data. Existing methods of generating and analyzing traces suffer from a variety of limitations including complexity, inaccuracy, lack of system references, short length, inflexibility, or applicability only to CISC machines.

We have built a system for generating and analyzing traces that addresses each of these problems. Trace generation is based on link-time code modification that makes the generation of a new trace easy. The slowdown of trace-linked code is small enough to allow reasonably accurate traces of user/system interaction. The system is flexible enough to allow great control of what is traced and when it is traced. On-the-fly analysis removes most limitations on the length of traces. The system is designed for use on RISC machines.

In this paper we describe the implementation of trace generation and on-the-fly analysis. We then review preliminary results from the analysis of user traces containing many billions of memory references. Very long traces are both useful and necessary for understanding the behavior of large, fast systems. We experiment with overall size, block size, and associativity in second level caches.

Copyright © 1989
Digital Equipment Corporation

1. Introduction

1.1. Background

For 20 years traces of memory access patterns have provided a window into program execution, allowing the simulation of memory systems with the goal of evaluating different cache designs [14]. The analysis of cache designs is becoming even more crucial as caches become dramatically faster than main memory and cache misses are an ever more important factor in system performance. For example, on today's RISC machines, it typically takes ten times longer to retrieve data from main memory than from the cache, whereas on machines currently being designed the ratio will be 100 to 1. Many new processor designs use RISC technology with very fast on-chip caches. Since it is infeasible to put sufficiently large caches on chip, somewhat slower, but very large, off-chip second level caches will be used to improve performance. Second Level caches of 16 megabytes or more may sit between the processor and a gigabyte of main memory.

Simulation is the primary tool for determining the appropriate sizes and characteristics of memory systems for the new RISC machines. To be able to accurately simulate the behavior of very large caches, very long RISC traces are needed. We are unaware of any existing traces that represent more that O(10 million) memory references. Existing traces are usually sufficient for the simulation of small caches on CISC machines, but they are too short to simulate multi-megabyte caches. We also believe that RISC traces are sufficiently different from CISC traces to warrant the generation of fresh traces. RISC code for a program is often twice as large as corresponding CISC code, increasing the number and range of instruction references. Also, effective use of the large register sets built into RISC machines reduces the number of data references compared to code for a CISC machine. At the very least, the balance of instruction and data references will change markedly.

Unfortunately, existing methods are inappropriate for generating long traces and for use on RISC machines. The most common software method involves the simulation of a program's execution to record all of its instruction and data references. This method is both slow and limited. Simulation is slow for CISC programs and may be slower for RISC programs because they contain more instructions each requiring a pass through the main simulator loop. A 1000x or more slowdown makes traces of real-time behavior, including kernel and multiprogrammed execution, impossible to accurately simulate. Some hardware methods spy on address lines to trace execution in real time, but these usually have limited capacity. In ATUM [1], trace generation involves microcode modification. The microcode for a machine is modified to trap address references and generate trace data. A microcode-modified machine runs only 10 times slower than an untraced machine. The method is not applicable to RISC machines since there is no microcode to be modified.

An additional problem with most existing methods is that they require the generation and storage of entire traces for later analysis. The requirement that traces be retained limits the length of the trace. The storage of many traces consisting of tens or hundreds of gigabytes of data is at best very expensive and at worst infeasible. The TRAPEDS [16] technique handles this by doing analysis on-the-fly within the traced program. Though TRAPEDS works well for multiprocessor tracing of single users, it does not trace operating system code or multiple users running in separate address spaces.

At WRL, where we design, build, and use high performance RISC machines, we have developed a method of trace generation that is appropriate for use on those machines. We make use of link-time code modification to generate code that will create a record of data and instruction references. Both user and operating system execution can be traced with 8 to 12x slowdown. On-the-fly trace analysis, in which trace data is analyzed as it is collected, eliminates the need to store traces and makes the analysis of very long traces feasible.

1.2. Goals and Status

The primary goal of this project is to implement a set of tools for trace generation on RISC machines, and to use them to analyze the memory system designs for our next machine. We require that:

- The traces must be complete. They must represent kernel and multiple users as they execute on a real machine. The memory references must be interleaved as they are during execution rather than being artificially interleaved separate traces.
- Traces must be accurate. The mechanism used must not slow down execution to the extent that the behavior of the system is no longer realistic.
- Tracing must be flexible. We must be able to pick and choose the processes to be traced, optionally trace kernel execution, and turn tracing on and off at any time.
- The traces must be long enough to make possible the realistic simulation of very large caches.

A new trace generation mechanism can satisfy the first three requirements, but new analysis techniques are required to meet the fourth. Since the trace lengths we require can outstrip storage capacity, analysis that is done off-line against stored traces is unacceptable. Therefore our research extends into methods of managing and analyzing long traces.

Initially, the traces will be used to analyze the behavior of multi-level cache hierarchies in very fast machines. Eventually, we will be interested in exploring other uses for the traces we generate. For example, Sites [1, 13] has had considerable success in using his traces as a debugging tool that gives a much more detailed picture of program execution than does profiling information. McFarling [10] discusses the use of profiling information to place code for cache optimization. Traces may also be a useful tool for determining code placement.

This paper describes an ongoing project. The first portion presents the design of the trace generation mechanism and on-the-fly analysis. At the time of this report, user tracing with on-the-fly analysis works extremely well. Kernel traces can be generated but do not yet work with on-the-fly analysis. The paper proposes solutions to problems still in the way of accurate kernel tracing. In later sections, we review the results of preliminary experiments tracing single and multiple user processes. The experiments validate our claim that long traces are both useful and necessary to understand the behavior of large multi-level cache systems. They provide insight into different mapping schemes, the usefulness of associativity, and effects of changing block size and overall cache size.

2. Trace Generation

2.1. Environment: The Titan

The original RISC machine designed and built at WRL was the Titan. It is currently our primary workhorse and the machine whose execution we have traced. The Titan is an experimental, high performance, 32-bit RISC workstation with a 45ns cycle time, different register sets for user and kernel, and a maximum memory configuration of 128 megabytes. A program may reference 64 general purpose registers. Titans run a modified version of Ultrix™ called Tunix (Titan Unix®) in which all process and memory management functions have been rewritten.

2.2. Link-time Code Modification

Our trace generation method relies on the ability to do link-time code modification. All compilers at WRL translate programs into an intermediate language, Mahler, which defines the abstract machine, hiding the details of the hardware from the compilers. The Mahler implementation [17, 18] consists of a translator and an extended linker. Object modules produced by the Mahler translator contain sufficient supplementary information to allow the linker to do global register allocation and pipeline scheduling, as well as the code modification we require for trace generation. In particular, basic blocks and their sizes are identifiable at link-time.

An option to the linker causes branches to trace code to be inserted in the program wherever a referenced address is to be recorded. A return address is saved in a reserved register, thereby avoiding a complex call/return mechanism. Trace code uses reserved registers to access its arguments so that memory accesses are minimized. When executed, the trace code computes a virtual address (physical address for the kernel) and writes the address in a *trace buffer*. An instruction address computed and written into the trace buffer is the address at which the instruction would have been located had the code not been expanded with trace branches. A trace resulting from the execution of trace-linked code represents an execution of normally linked code. An example of the effect of code expansion follows:

Original code:

```
(addr1)   lab1: r1 := 4[r2]
                r1 := r1 + 1
                8[r2] := r1
                if r1>0 goto lab2
                null
(addr2)         r1 := r9 + r10
                ...
```

Expanded Code (symbolic):

```
lab1: call InstrTrace(5,addr1)
      call LoadTrace(ADDR(4[r2]))
      r1 := 4[r2]
      r1 := r1 + 1
      call StoreTrace(ADDR(8[r2]))
      8[r2] := r1
      if r1>0 goto lab2
      null
      call InstrTrace(7,addr2)
      r1 := r9 + r10
      ...
```

where **ADDR(d[rn])** means the address in memory described by displacement **d** and the contents of base register **rn**.

Expanded Code :

```
lab1: r48 := pc, goto InstrTrace
      r49 := 5
      .word (addr1)
      r48 := pc, goto LoadTrace
      r50 := loadaddr 4[r2]
      r1 := 4[r2]
      r1 := r1 + 1
      r48 := pc, goto StoreTrace
      r50 := loadaddr 8[r2]
      8[r2] := r1
      if r1>0 goto lab2
      null
      r48 := pc, goto InstrTrace
      r49 := 7
      .word (addr2)
      r1 := r9 + r10
      ...
```

A Titan references memory addresses only during load and store instructions and instruction fetches. The linker inserts a branch to trace code at each load, each store, and at the beginning of every basic block. A basic block is a sequence of instructions with a single entry and a single exit. The trace code for loads and stores inserts into a trace buffer the address to be referenced together with a bit indicating whether the reference was a load or store. The trace code for basic blocks inserts the instruction address as well as the size of the basic block. Since all of the instructions in a basic block are executed if the first one is, a single trace entry is sufficient to later simulate the correct sequence of instruction fetches. The only information absent in the current implementation is the interleaving of loads and stores with instruction fetches within each basic block. The information is obtainable, but to keep traces compact and allow longer uninterrupted runs before the trace buffer fills, we do not keep it.

2.3. Operating System Support

The operating system's primary trace-related job is the management of the trace buffer. The trace must reflect the actual sequence of memory references made by a mixture of user and kernel programs. Therefore, the trace buffer is shared among the kernel and all running processes. Only those processes that have been linked for tracing actually use the buffer.

At boot time, the trace buffer is allocated from the free page pool. The current system runs on a Titan with 128 megabytes of memory. The trace pages are permanently associated with the trace buffer and are not pageable. During execution, the machine behaves as if it had less physical memory. We have experimented with 32 and 64 megabyte trace buffers.

From within the operating system, the buffer is referenced directly using its physical addresses. To make user references to the trace buffer sufficiently fast, the buffer is mapped into the high end of every user's virtual address space, as shown in Figure 1.

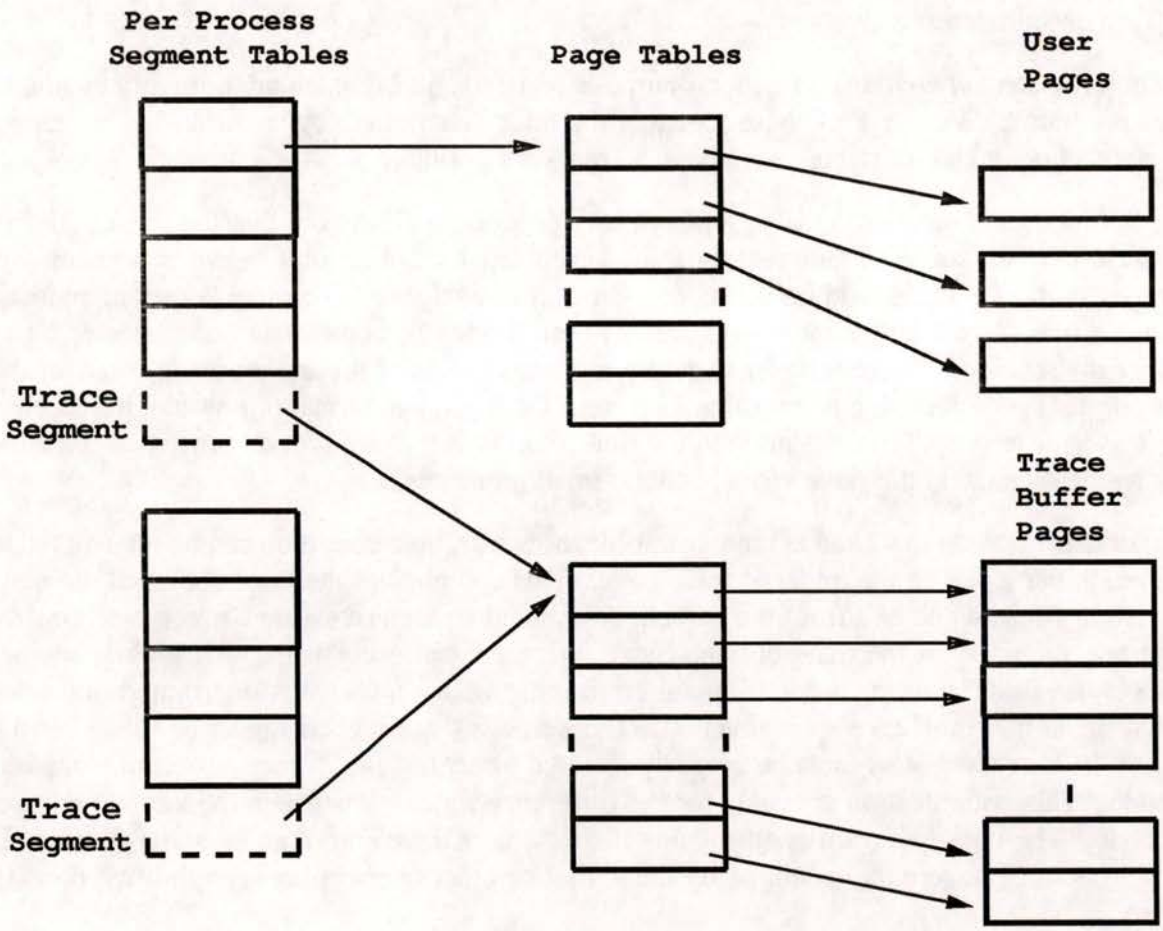


Figure 1: Mapping of the shared trace buffer

This mapping allows the user trace code to write directly to the trace buffer, referencing it by its virtual address. The Tunix implementation of a process's virtual memory as a variable sized set of segments, each associated with a page table, makes this particularly easy. By default, each

process has text, initialized data, uninitialized data and stack segments. The trace pages make up a shared fifth segment. The stack, which usually resides at the high end of the 1 gigabyte virtual address space, is moved to below the trace buffer. The address space is large enough that this does not constrain user programs. Since a single trace segment with its page table is set up at boot time to be shared by all processes, the operations required on creation of a new process are minimal, involving only the copy of a pointer to the segment.

2.4. The Trace Code

For efficiency, the trace code itself was written in TASM (Titan Assembly Language) using registers to avoid memory references. The only memory references in the trace code are writes to the trace buffer. All other values needed to manage tracing (pointer to the next available trace location, value to be written, return address, trace flag, etc.) are kept in registers. A simple branch instruction is used to return from the trace code. In user programs, 18 TASM instructions are executed to generate a data trace and 22 to generate an instruction trace. In the kernel, 14 and 18 are executed, respectively.

A Titan register set contains 64 general purpose registers, 58 of which are normally available to user programs. We use 8 of these registers for trace generation. As a result, the operating system and code linked for tracing each have 50 registers available.

The kernel uses a separate register set from user processes. To assure that the trace registers are always correct, some of their values must be copied back and forth between register sets when moving between user and kernel mode. On every transfer from one register set to another, two values (the current buffer index and buffer pointer) must be copied from one register set to the other via memory. If the transfer is between a kernel register set and a user register set the buffer pointer is not copied, but is recalculated from the index and virtual or physical buffer base address as appropriate. The translation to a virtual address is the same for all processes since the buffer resides at exactly the same virtual address in all processes.

Kernel execution on the Titan is uninterruptible, however, user execution can be interrupted at any time, in particular in the midst of trace code. This complicates the trace code and the code executed on entry to and exit from the kernel. Additional code must assure correct synchronization among the users of the trace buffer. Since interrupts can occur between arbitrary instructions, a lock value (requiring a register) is used to indicate that a user was interrupted mid-trace and that the kernel must take special action. The kernel's trace registers must be made consistent, and the user's registers must be properly restored to continue mid-trace on return from kernel mode. This complication accounts for the difference in length between the kernel and user trace code. The only inelegant result of this is that a user trace entry can be split, with an arbitrary amount of trace data generated by the kernel or other users in between the two parts of the entry.

2.5. Controlling Tracing

The set of programs traced is determined by which ones are specially linked. They may run side by side with programs that are not traced. The kernel is traced only if it is trace-linked.

Tracing is done only when a trace flag kept in one of the trace registers is turned on. Before writing to the trace buffer, the value of the trace flag if checked. If it is off, control returns to the program without writing to the trace buffer. Tracing is turned on at the first interrupt following a write to a location in */dev/kmem*, a UNIX device corresponding to kernel memory that allows privileged users to modify kernel values. Tracing is turned off in an analogous fashion.

When tracing is off, 4 to 6 additional instructions are executed at every trace point. No additional cost is incurred during the execution of user code that is not linked for tracing whether or not tracing is turned on. The modified kernel, even when not linked for tracing, is slower by a factor of about 2 because even a non-traced kernel must do extra work on every kernel entry to assure that the trace registers are kept consistent.

2.6. Explicit Trace Buffer Entries

In addition to the automatic generation of address reference information as the result of linker inserted branches, it is possible to explicitly call trace routines from either the user or the kernel to insert arbitrary informative items into the trace. Currently we make use of this in the kernel.

The addresses inserted in the trace buffer by the kernel are physical addresses, whereas those inserted by user processes are virtual addresses. It is essential for the analysis code to be able to tell one from the other. It is also useful to be able to associate sequences of virtual addresses with a particular process when more than one is being traced. Therefore, on every transfer into or out of the kernel a change mode entry is made in the trace buffer. The entry indicates whether the change is from user to kernel or kernel to user and which user process is involved.

2.7. Format of Trace Buffer Entries

Entries in an address trace can be divided into two types based on whether they fill one or two 32-bit words of the trace buffer. Data reference entries, the majority of all entries, are one word long. All others are two words long. An address generated in kernel mode is physical. An address generated in user mode is virtual.

Data Load/Store entries take up a single 32-bit word. Since we trace word addresses from a 1 gigabyte address space, the two high order bits are always available as a type field. Any trace entry with non-zero high order bits is a data entry. The first word of other types of entries contain type information in the high order byte.

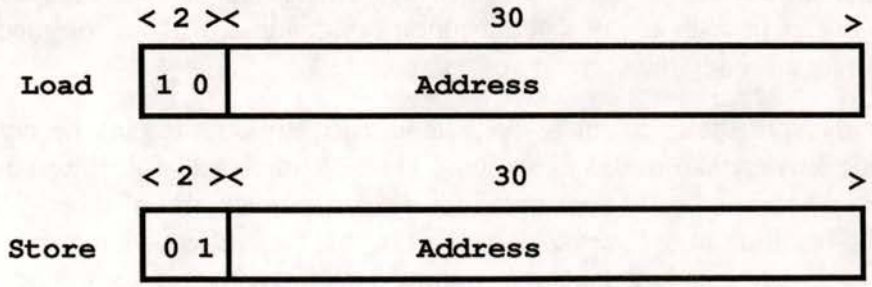


Figure 2: Data entry format

An Instruction entry represents the beginning of a basic block. The essential characteristic of a basic block is that all instructions in the block are presumed to execute sequentially.

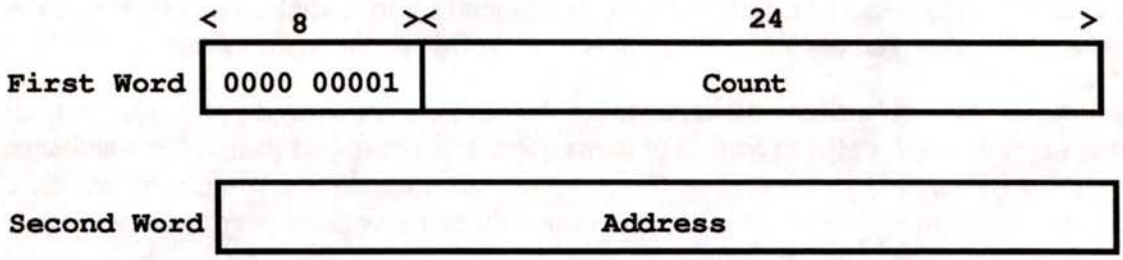


Figure 3: Instruction entry format

Count is the number of instructions in the basic block starting at the address in the second word.

2.8. Kernel Tracing: A Few Special Considerations

Our goal is to be able to relink the kernel just as we do a user program in order to trace its execution. This is possible for nearly all kernel code, with the exception of a portion written in TASM and never modified by the linker. This section corresponds to the very low level machine-dependent code in Unix and is currently not traced. In Tunix, the code determines the cause of traps and interrupts, saves and restores coprocessor state and process state (register contents), manipulates the clock and translation look aside buffer (TLB), and manages returns from interrupts. It also includes the idle loop. We will eventually want to trace the TASM code. To do this, calls to trace routines will be selectively inserted into the code (all but the idle loop) by hand.

3. Trace Management

3.1. Extracting the Traces from Memory

The system as described thus far can very quickly gather an address trace whose length is the size of the trace buffer. Tracing slows execution by about 8 to 12 times. Even with a 64 megabyte trace buffer (half of the Titan's memory), the trace represents only about 2 seconds of untraced Titan execution time or 30 to 35 million memory references. While such traces are much longer than those commonly available and are interesting for some purposes, they are not nearly long enough to analyze the behavior of a very large cache. This means that the trace data must be extracted from the buffer in such a way that execution can continue without too much disruption, in particular, without affecting the accuracy of the trace data.

Neither extraction nor analysis of the data can be done simultaneously with tracing because either is orders of magnitude slower than trace generation. Thus, all methods of dealing with long traces require that tracing be periodically interrupted for a significant amount of time. The challenge is to assure that the resulting traces are *seamless*, that is, that they reflect address reference patterns that would have occurred had the machine continued tracing without interruption.

The interruption may entail extracting the partial trace and writing it, possibly in a compressed form, to some storage medium for later analysis. Alternatively, the partial trace might be

analyzed immediately, eliminating the need to save the trace. The first possibility does little to solve the ultimate problem of very long traces where storage is impractical or impossible. Tracing of 5 minutes of Titan execution could generate nearly 10 gigabytes of uncompressed trace data.

While we have implemented a mechanism to extract, compress, and write partial traces to high density tape, our preferred method is to analyze the trace data on the machine being traced as it is generated. When the trace buffer becomes nearly full, the operating system turns off tracing and runs a very high priority analysis process. As with any other process running on the traced machine, the trace buffer is mapped directly into the analysis program's address space so that the data can be directly accessed.

Execution of the analysis program is controlled by the use of a *read* system call on a special file. The program opens the file and attempts to read it. The *read* returns the virtual address of the beginning of the trace buffer and the number of available entries only when the buffer is full (or nearly full). The program may then do anything it chooses with the trace data. The operating system guarantees that during the execution of the analysis program, tracing is turned off and traced programs are not scheduled for execution.

When all current data in the buffer has been processed, the special *read* is once again executed, tracing is turned back on, and traced user programs can once again execute.

3.2. Reproducibility of Results

A valuable characteristic of stored traces is the reproducibility of simulation results. When traces are generated and stored for later use, the identical trace can be used to simulate and compare different cache organizations. On-the-fly analysis does not provide that capability. While sub-traces of executions of any single deterministic user program will be identical, the inclusion of multiple processes will cause traces to vary from one run to the next. Likewise, kernel traces will never be identical from run to run.

There are a couple of potential solutions to this problem. First, it might be possible to simulate more than one cache organization at a time. The utility of this approach is clearly limited because it does not allow old results to be compared with new ones. Another possibility is to do a sufficient number of controlled runs to determine the variance in results for each cache organization so that any statistically significant difference between two different caches organizations can be determined. While this is more painful and time consuming than either stored traces or multiple simultaneous simulation, the results will more accurately represent the reality of execution on an actual machine.

4. Accuracy of the Trace Data

An accurate address trace represents the actual run-time sequence of memory references made by a set of trace-linked programs. There are a number of ways in which the trace data can be distorted so that it does not accurately reflect untraced execution. These result either from not tracing code that would normally be executed, or from tracing extra code that is executed only in a traced system, or from tracing code that is executed at a different time in a traced system. Factors affecting the distortion are the design of the tracing mechanism, the slowdown caused by

tracing, and the interruption of traced programs in order to execute trace extraction and analysis code.

4.1. Eliminating Gaps for Seamless Traces

For a trace to accurately represent real execution, there must be no gaps in the trace. One cause of such gaps can be the interruption in tracing during trace extraction and analysis.

If only user programs that are not time dependent are traced, seamlessness is not an issue. Traced users are excluded from execution while the analysis program is run and resume execution at precisely the point they were stopped once the trace buffer has been emptied. Since the cache simulation analysis we do takes approximately two orders of magnitude more time than does execution of the traced code, time dependent programs will behave very strangely, if they work at all.

The effect of analysis breaks is more difficult to deal with when the kernel is traced. The kernel cannot be prevented from executing while the analysis program runs and tracing is turned off. While it is not appropriate to trace kernel operations that are executed on behalf of the analysis program, the trace should include code that services interrupts belonging to the traced process, such as the completion of I/O initiated by a traced user process.

Careful placement of the code that turns off tracing prior to running the analysis program assures that the trace sequence is a realistic one that could have actually occurred in the absence of tracing. We have not yet come up with a solution to missing interrupt traces. For the moment, we will analyze the problem to understand how often this happens and to determine the effect on the accuracy of the analytical results.

4.2. Trace-Induced TLB Faults

The second type of inaccuracy occurs when code is traced that is solely an artifact of tracing. Since the only extra code executed in user mode is the trace code itself, which is not traced, this inaccuracy occurs only during kernel tracing.

For example, a TLB fault that would not occur during the execution of untraced code is generated whenever a user process first references a new page of the trace buffer. Since TLB faults are handled in software in the kernel, a kernel trace would contain a record of the execution of the TLB fault handler. If the trace of the fault handler were sufficiently long, the next user reference to the trace buffer could be to the next page of the buffer and could generate another TLB fault. Execution would thrash in a nasty loop filling the trace buffer only with fault execution traces. This situation did occur during the early debugging of the system. Note that references to the trace buffer never cause page faults because the buffer is locked into memory.

To solve this problem, it is not sufficient to turn off tracing whenever the user's trace code is entered because other types of interrupts, which should be traced, are very likely to occur during the tracing code. After all, one is eight times as likely to be in the middle of tracing code as in regular code when an interrupt occurs. It is also possible for multiple interrupts to be handled on a single trip into the kernel. For example, an I/O interrupt could occur while a TLB fault on the trace buffer was being handled. One would like to ignore the TLB fault but trace the handling of

the I/O interrupt. Our solution is to modify the low level TASM interrupt code in the kernel to detect this situation and turn off tracing whenever a TLB fault on an address in the range known to be the trace buffer occurred, and to turn it on again on return to user mode or if additional work is to be done prior to returning to user mode.

Another cause of trace induced TLB faults is the modification of the tlb during trace extraction and analysis. Execution of the analysis code and references to many trace buffer pages are guaranteed to substantially modify the TLB. After the analysis program has run, the traced user process would normally have to re-fault in its relevant translations. To solve this, we may save the contents of the TLB after turning off tracing but before running the analysis program. The contents of the TLB are then restored prior to turning tracing back on.

4.3. Interrupt Timing

Traces may not accurately represent normal execution because code segments are executed at a different time, i.e. in a different order, when tracing is on. Any slowdown in execution means that various interrupts happen sooner or more frequently than they would without tracing. This will affect traces for user code that does asynchronous I/O or relies on signals. I/O will appear to be considerably faster because less user code (and more trace code) is executed between initiation and completion of an I/O operation. We consider this distortion a necessary evil.

Also problematic are the additional timer interrupts that are not only traced (in kernel mode) but affect the rate at which process switching is mandated. This effects the scheduling of user processes because less code will be executed before a single process's time quantum runs out. This can be fixed by changing the quantum associated with traced processes. If the effect of additional timer interrupts in kernel traces turns out to be problematic, it will have to be accounted for in the analysis program rather than by modifying the traces.

4.4. Process Switch Interval

The decreased execution speed for traced processes influences the accuracy of multiple process traces. Since the process switch interval is measured in time rather than instructions, the number of original instructions executed between process switches is 8-12 times less than normal. If such traces are used to simulate cache behavior, the resulting performance will be erroneously low. To account for this we increased the process switch interval. This was done without changing the interval at which clock interrupts were serviced to check for external interrupts since interrupts still need to be recognized quickly.

Initially, we increased the process switch interval by a factor of 8 to reflect actual switching on a Titan. This corresponds to the execution of approximately 200,000 user instructions between switches. Since we are interested in simulating machines that are faster than the Titan we also experimented with an interval corresponding to 400,000 instructions between switches. We do not know of any other multiprocess trace generation system which attempts to take this into account.

The multiprocess results presented later in the paper were all run with a 400,000 instruction maximum interval. The actual average number of instructions executed between process switches for our multiprocess benchmark is about 175,000 in part because the quantum is

measured in cycles rather than in instructions. With no stalls, the Titan would issue one instruction per cycle, but it does not run at that rate. Also, only compute bound programs regularly use their entire time quantum on each switch.

5. Verification

It is difficult to verify the absolute correctness of many gigabytes of trace data, however, we were able to use a number of techniques to convince ourselves of their accuracy.

5.1. Comparison with Simulation Results

An instruction level simulator existed for the Titan. The simulator already produced cache hit and miss data and was modified to produce traces for short, but real, user programs. For ease of comparison, our regular cache analysis program was modified to write out traces in exactly the form produced by the simulator. The resulting trace files were compared and the number of cache hits and misses were compared.

The differences we noted first exposed bugs and then showed some surprising cache performance results that would not have been obvious using only one trace method. While instruction addresses matched, we were puzzled by differences in data addresses and data cache hit rates. It turned out that the simulator made different assumptions about the virtual address at which the user process's stack began from that made by the operating system. This affected the cache locations used by stack references. The surprising result was that this small change resulted in as much as a 15% difference in the hit rate in the data cache.

While the simulator could not be used to simulate kernel details, it was still extremely useful. We were able to verify that the trace sequence was correct and that the addresses computed in the trace code corresponded to the correct addresses referenced by executing code that was not expanded for tracing. Since the link-time modification is the same for user and kernel code, we believe that this justifies confidence in the accuracy of both kernel and user traces.

5.2. Checking for Gaps

In addition to checking the correctness of the information created by the trace code, it was necessary to assure that kernel and user accesses to the trace buffer were properly synchronized, and that no gaps in traces were produced when the analysis program ran.

Two techniques were used to verify correct synchronization. First, we traced a program that infinitely executed a very simple tight loop, thus generating an easily recognizable pattern, and then ran a variant of the analysis program that checked whether anything was missing or out of place.

Second, the trace entry generated by the operating system on entry to the kernel was extended to include a sequence number that was incremented on each kernel entry. Again, the analysis program checked that no entries into kernel mode were missed or overwritten in the trace buffer.

6. Trace Analysis

6.1. Panama: A Cache Analysis Program

Panama is our first and most general cache analysis program. The requirements were that it be flexible and easily modifiable so that cache characteristics could be changed from run to run. The program should take up as little space as possible so that we can run large user programs without additional paging overhead. Since analysis is the slowest part of the process taking easily 10 times as long as trace generation, it should be as fast as possible.

To achieve speed and small size, we sacrificed run-time flexibility and were satisfied with compile-time flexibility. Constants whose values are specified at compile-time define the following cache characteristics:

- Number of cache levels (1 or 2)
- Split or integrated data and instruction caches
- Degree of associativity
- Number of cache lines
- Line size
- Write policy (write through or write back)
- Cost of hit and miss (in units of processor cycles)

A version of the program is compiled for each cache configuration, thus minimizing space allocated for run-time data structures and eliminating code that is not executed for that configuration. Panama can also simultaneously simulate a fully associative version of the second level cache.

Output of the trace analysis is generated at intervals that can be based on the number of instructions, trace entries, or memory references encountered. The data written includes both interval and cumulative summaries of the following information:

- For each cache:
  - number of references
  - miss ratio
  - contribution to the cycles per instruction (CPI_{cache})
  - the percent of misses that resulted from user/kernel conflicts
- Contribution to the *CPI* by second level cache compulsory misses (first time references)
- Second level cache contribution to the CPI if fully associative
- For user mode and kernel mode:
  - Number of cycles executed and *CPI*
  - *CPI* broken down by instruction, load and store contributions

Panama works well, but remains a candidate for speed optimization. When simulating split instruction and data first level caches and a direct-mapped second level cache, analysis plus trace generation take about 100 times as long as untraced execution. For example, tracing and analyzing a run of TV, a WRL timing verification program, extends the run time from 25 minutes to 45 hours.

6.2. Tycho: Evaluating Associativity

Tycho is a cache simulation program developed by Mark Hill [6] that allows many different single level cache configurations to be simulated concurrently. The algorithm used is the special case of all-associativity simulation [9]. Although our primary interest is with multi-level caches, we used Tycho to determine whether very long traces produced different results than short traces and whether such effects varied with the degree of associativity.

6.3. Saving Traces for Later Analysis

On-the-fly analysis techniques allow us to analyze traces of unlimited length, but the need for long traces outside of WRL convinced us to generate and save a set of traces for distribution. Data storage was the primary problem to be overcome. We decided to put the data on tape for two reasons: capacity and portability. Sony® 8mm video cartridges used with an Exabyte™ drive chosen for their large storage capacity, 2 gigabytes, and their convenience. The cartridges are the size of a normal audio cassette tape.

Techniques similar to the Maching scheme used in [12] were used to pack massive amounts of trace data on each tape. The original trace data is first converted into a *cache difference trace* consisting of reference points and differences. Addresses are represented as the difference from a previous address, rather than as an absolute address. This *differencing* creates many repeated patterns in the trace reference string. The cache difference trace is then run through the Unix *compress* program. *Compress* uses Lempel-Ziv [20] compression to recognize common sequences of data and compress these sequences into single code words. Reference locality and the regularity of trace entry formats allows trace data to be very effectively compressed. We were able to reduce the storage requirements to between 0.38 and 3.0 bits per memory reference, with most traces in the 2-3 bit range. This allowed us to store a trace of about 8 billion memory references on a single tape.

7. Experiments with Single and Multiple Process User Traces

7.1. Base Case Cache Organization Assumptions

A number of assumptions have been made about the type of machine we are interested in analyzing. They are based on discussions with WRL's hardware engineers and represent some broadly accepted projections for workstations of the future. The assumptions define the basic cache organization whose characteristics we then varied and simulated.

The only assumption related to the generation of traces is that of a RISC architecture in which instruction fetches and explicit data loads and stores are the only forms of memory reference. This makes the simple form of the traces useful even though they contain no information about

instruction types other than loads and stores. The remaining assumptions are built into the Panama analysis program.

We assume a pipelined architecture where, in the absence of cache misses, a new instruction is started on every cycle. This is possible when the page offset is simultaneously used to retrieve the real page from the TLB and to index the correct line in the cache. Since we do not consider delays other than those caused by cache misses, we count the base cost of instruction as one cycle. The goal of the memory hierarchy design is to keep the *CPI* as close to 1 as possible.

We also assume the machine is fast enough to require two levels of cache. The additional cost of going to the second level cache, in case of a miss in the first level cache, is assumed to be 12 cycles. This is the same as the cost of going to main memory on a Titan. The cost of going to main memory is assumed to be 200-250 cycles depending on the size of block that is retrieved.

The first level data cache is presumed to be a write through cache with an associated write buffer. In our experiments, the capacity of the write buffer is 4 entries. If there is a entry available, the write occurs at no additional cost, otherwise the write stalls. An entry is retired, that is, it commits in the second level cache and frees space in the write buffer, every 6 cycles. In other words the write buffer behaves like a 4-entry FIFO, queuing up the writes to the second level cache.

The second level cache is write back. It may be either virtually or physically addressed. We have used a mapping based on virtual user addresses and physical kernel addresses. It is described in detail in Section 7.5.

7.2. Choice of User Programs

Our choice of user programs to trace is based upon the assumption that in the future machines will have much larger memories and that programs will be written to use that memory. Ideally, the programs should characterize the average workload of a future system. Unfortunately, an "average" workload is difficult to define. Not only are there wide variations in use from one user to another, but the usage changes with time. Prediction of a future workload is a difficult problem. We are confident of one thing. Applications will grow to use the large amounts of memory available in future machines.

Most programs today have working sets of less than 16 megabytes. Some programs have large address spaces but only use very small portions at a time to avoid thrashing on small memory machines. Nearly every program we have chosen to trace is real and currently in use on existing large machines. Many are memory hogs by today's standards, but not simply because they are poorly written. They require large amounts of memory because the problems they are solving are inherently large.

Figure 4 describes the programs we have traced thus far. In the remaining sections we will focus on the following four examples.

- **TV** is a timing verifier for VLSI circuits which is in regular use at WRL. It generates a linked data structure and then traverses it. It was written in Pascal by Norm Jouppi. The program is traced while analyzing the MultiTitan CPU, which has 180,000 transistors. For this run it requires 96 Mbytes of memory.

Traced User Programs	
Name	Description
Make	Unix program to maintain, update, and regenerate groups of programs. The make program was used to generate a sequence of compiles and loads. It made calls to *cc*, *rm*, and *cat*.
Cc	C compiler front end. It initiates the C pre-processor, C compiler, Mahler compiler, and loader.
Cpp	C language preprocessor.
Ccom	First phase of C compilation on the Titan.
Mc	Titan Mahler [18] intermediate language compiler.
Xld	Titan extended linker and loader.
Cat	Unix utility to concatenate files.
Cp	Unix utility to copy files.
Vi/Ex	Unix text editor.
Ps	Unix utility to read process status.
Ls	Unix utility to list directory contents.
rm	Unix utility to remove a file.
Tcsh	Unix shell program.
Magic	VLSI editor [11]. It has a number of features not found in other editors, including design rule checking, plowing, etc.
Grr	Printed circuit board router [4].
TV	Circuit timing verifier [7].
SOR	Fortran implementation of the successive overrelaxation algorithm [3, 8] that uses sparse matrices.
Linear	Program to solve linear systems of equations using sparse matrices [15].
Tree	Compiled Scheme program which builds a tree data structure and searches for the largest element in the tree [2].

Figure 4: Descriptions of programs used in trace experiments

- **SOR** executes Renato De Leone's implementation of the successive overrelaxation algorithm using sparse matrices. It is written in Fortran with static array allocation. It operates on a 800,000 by 200,000 sparse matrix with approximately 4 million (0.0025%) non-zero entries and takes 62 Mbytes.
- **Tree** is a compiled Scheme program, written by Joel Bartlett, which builds a tree data structure and searches for the largest element in the tree. Bartlett's implementation of Scheme uses a garbage collection algorithm in which memory space is split into two halves. When no memory is left in one of the halves, data is compacted into the empty half. It uses 64Mbytes of memory.

- **Mult** is a multiprogramming workload consisting of:
  - a *make* run compiling (from C) portions of the *Magic* source code
  - *grr* routing the DECstation 3100 printed circuit board
  - *Magic* design rule checking the MultiTitan CPU chip
  - *Tree* on a smaller problem given 10Mbytes of working space
  - another *make* run that calls *xld* on the *Magic* code
  - an infinitely looping shell of interactive commands (*cp, cat, ex, rm, ps -aux,* and *ls -l /**)

This set of programs uses approximately 75 Mbytes of memory.

User Program Trace Information				
Name	References (billions)	Instructions (% of refs)	Loads (% of Instrs)	Stores (% of Instrs)
Tree	5.4	67.2%	30.3%	18.6%
Tv	17.1	72.9%	26.8%	10.4%
Sor	9.6	73.2%	28.1%	8.2%
Mult	7.6	68.5%	30.6%	15.4%

Figure 5: User trace characteristics

Figure 4 contains the characteristics of the example traces. Not surprisingly, the number of instructions as a percent of memory references is higher than in CISC programs. The average percent of references that are instructions is 70.5%, whereas the average value for traces examined in [6] was 53.3%. All of the programs had been compiled with all available optimization including global register allocation [17].

7.3. *CPI* as a Measure of Cache Behavior

The most common measure of cache performance is the *miss ratio*, the fraction of requests that are not satisfied by a cache. Though the miss ratio is useful for comparing individual caches, it does not give a clear picture of the effect of a cache's performance on the performance of the machine as a whole, or of the relative importance of the performance of each cache in a multi-cache system. For example, one cache may have a much higher miss ratio than another, but be used much less frequently and result in less overall performance degradation than the first.

A better measure of performance in a multi-cache system is *CPI*, the average *cycles per instruction*. *CPI* accounts for variations in the number of references to the different caches and differences in the costs of misses to the caches, and can be broken down into the components contributed by each part of the memory hierarchy. We are concerned only with memory delays and ignore all other forms of pipeline stalls, assuming one instruction is issued per cycle. Therefore, for our base machine the *total CPI* over some execution interval is:

$$CPI_{total} = 1 + CPI_{data} + CPI_{inst} + CPI_{level2} + CPI_{wtbuffer}$$

where $CPI_{cache\text{-}type}$ is the contribution of a cache and $CPI_{wtbuffer}$ is the contribution of the write buffer.

If for each cache,

$m_{cache\text{-}type}$ = number of misses during the interval

$c_{cache\text{-}type}$ = cost in cycles of a miss

i = the number of instructions in the interval

then

$$CPI_{cache\text{-}type} = (m_{cache\text{-}type} \times c_{cache\text{-}type})/i$$

The goal is to minimize the total *CPI*.

Admittedly, *CPI* is an architecturally dependent measure, but machine designers are most often interested in the performance of a specific architecture rather than cache performance in a vacuum. CPI_{total} tells much more about the performance of our base architecture than do the miss ratios of the individual caches.

For example, the top graph in Figure 6 shows the miss ratios for the three caches during a 3 billion instruction segment of a run of the Mult set of programs. The bottom graph shows the total *CPI* and *CPI* contribution of each cache for the same run. While the miss ratio for the data cache usually exceeds that of the instruction and second level caches, its actual contribution to the total *CPI* is much smaller than that of the instruction or second level caches. The *CPI* graph also shows that the second level cache often contributes nearly twice as much to the overall performance degradation as do either the instruction or data caches.

In the rest of this paper, we will present trace results in terms of two different measures of *CPI*, the *interval CPI* and the *cumulative CPI*. The *interval CPI* is measured at regular intervals over the course of a trace and simulation run and at each point is the average for the previous interval. The *cumulative CPI* is the CPI averaged over the entire preceding portion of a run.

7.4. Long Traces are Nice and Necessary

Some of the earliest data collected validated our belief that long traces were necessary to understand the behavior of large caches. Each point in Figure 6 represents an average over 100 million instructions, or approximately 150 million memory references. Most existing traces are less than 10 million references long. Each point in Figure 6 is ten times that. The variation in the CPI_{total} over the length of the trace makes clear the necessity of long traces.

A more dramatic example is shown in Figure 7. The graphs represent a run of TV that executed for 12.5 billion instructions. The graphs show the data for 1.5 billion instructions, from instruction 10.6 billion to 12.1 billion. Each point on the graph represents an average over the previous 10 million instructions, near the entire length of most available traces. Analysis done on only 10 million instructions could conclude that the CPI_{total} was anywhere between 1.7 and 6.8. Moreover, interesting behavior did not begin until after 9 billion (!) instructions had been executed. Up until that point, the graph was flat with a constant CPI_{total} of 1.7.

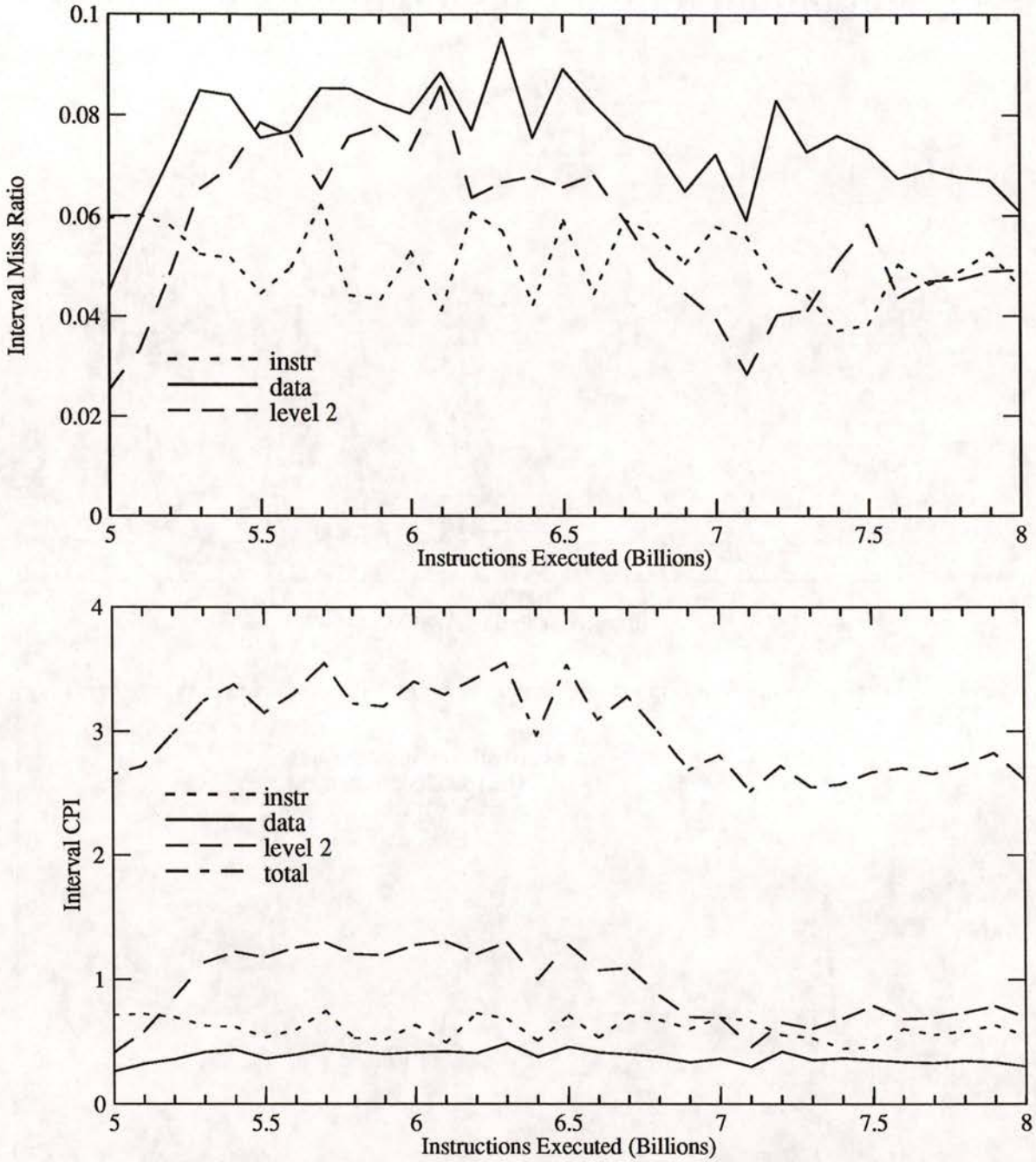


Figure 6: Interval miss ratios (top) and interval *CPI*s (bottom) for Mult

In the above examples, the *CPI*s and miss ratios were measured at intervals of 100 million instructions. The instruction and data caches are both 4K bytes consisting of 256 4-word lines. The second level cache is 16M bytes made up of 4096 32-word lines. All three are direct-mapped.

This example also illustrates how the behavior of each phase of the TV program is identifiable. The two major phases of the computation, max and min delay analysis, start about 1065 and 1140, respectively. Note the similarity in the graphs of CPI_{inst} for each phase. Each of eight subphases corresponding to the rising and falling of the four clock phases of the simulated chip are also clearly identifiable. The high points in the graphs of CPI_{data} and CPI_{level2} that match

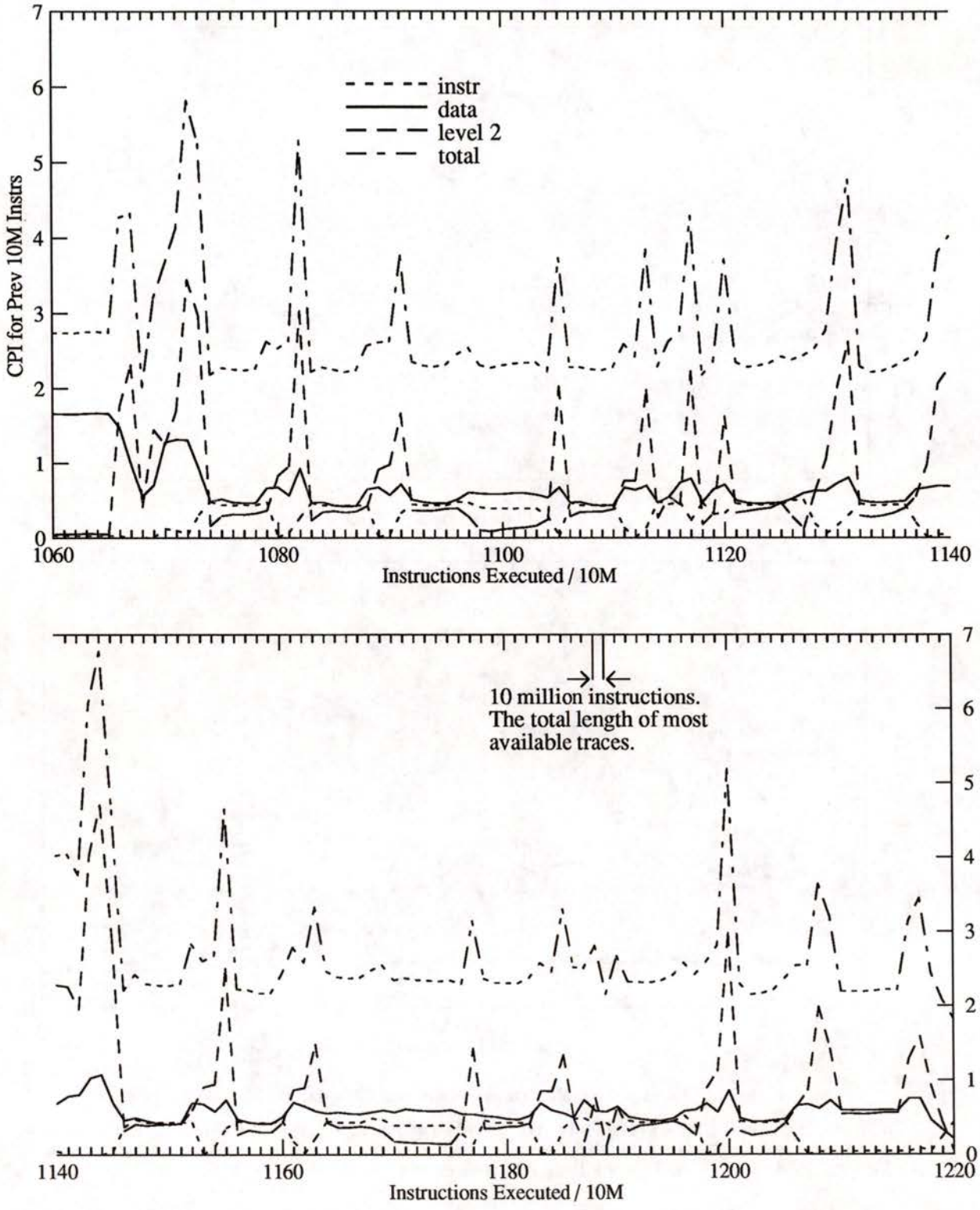


Figure 7: Interval *CPI*s for TV in two parts

This example used a 4K byte instruction cache with 64 byte lines, a 4K byte data cache with 32 byte lines, and a 16M byte second level cache with 256 byte lines.

lows in the graph of CPI_{inst} are loops that execute small sections of code and go through large volumes of data.

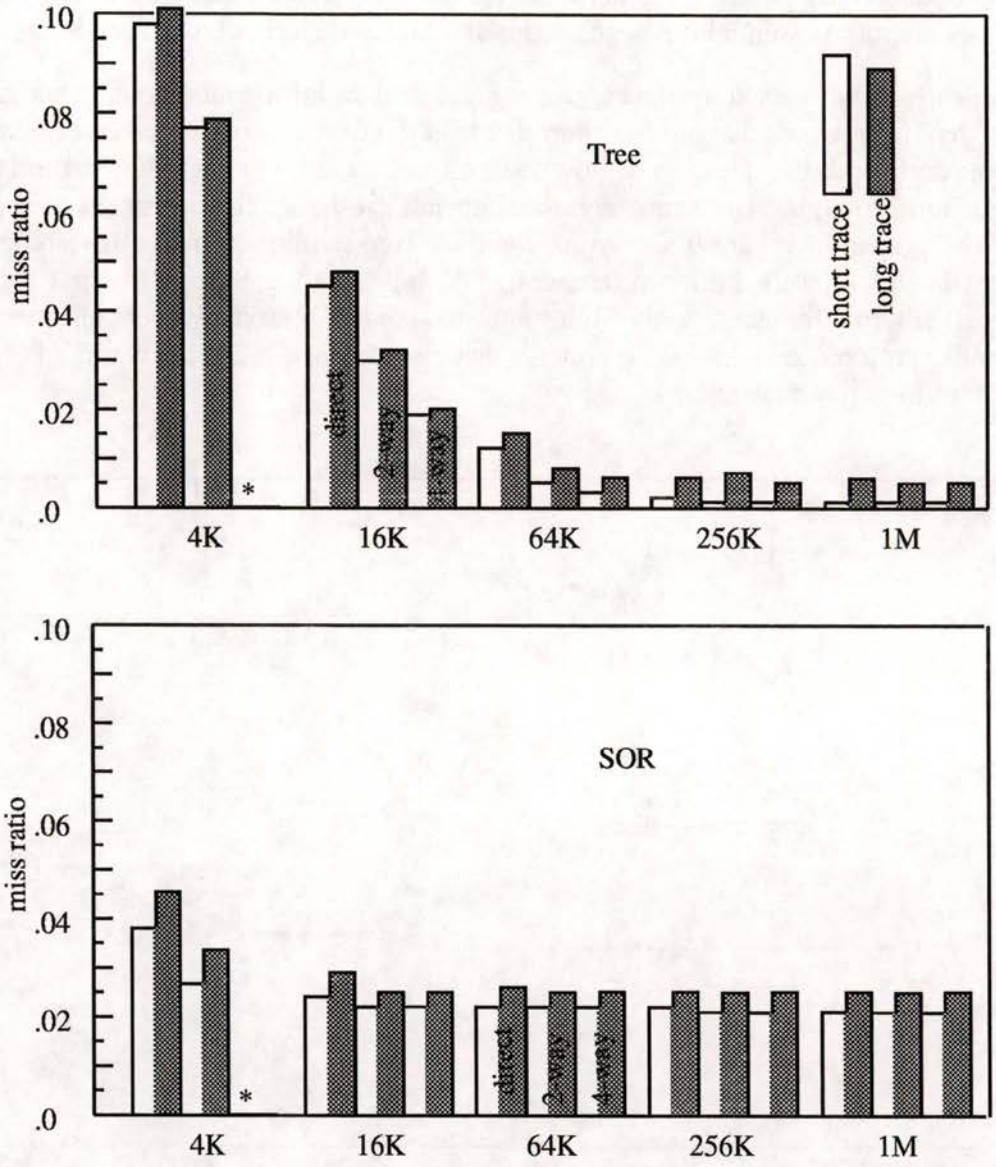


Figure 8: Miss ratios for short (500K refs) and long (150M refs) traces

Data was collected for 5 cache sizes, 4K to 1M. The data are for single level mixed instruction and data caches. * Note that there is no data for the 4K byte cache with 4-way associativity.

The usefulness of long traces is not limited to large multi-level caches. Using Tycho, we examined the effects of trace length on performance results for single level caches, varying cache size and the degree of associativity. Running Tycho on traces of increasing length showed that even the performance of programs on single level caches is not accurately modeled by short traces. We ran Tycho against two sets of traces generated by Tree and SOR. The first run used 500,000 references, the second more than 150,000,000. In the two cases shown in Figure 8, miss ratios dropped when analyzed for the longer trace. The results are not due simply to cold start

problems since Tycho accounts for them. The miss ratio will not always be lower for long traces. The difference depends on where in a long trace the short trace is chosen. Were we to run the same experiment with TV, choosing an early part of the run for the short trace, the miss ratio would undoubtedly go up. In general the results show that the differences are greater, and so long traces are more essential for large caches and higher degrees of associativity.

We have shown that very long traces give a great deal of information about a program's behavior and that using traces that are too short can lead to erroneous performance estimation. We have not yet dealt with the question of how long a trace must be to get valid performance data. How long is long enough? Unfortunately, that depends on the application or set of applications running. The graphs in Figure 9 show this clearly. Tree terminates before it stabilizes. SOR gradually stabilizes after 10 billion references. TV appears to stabilize early and then takes a jump after 10 billion references. Only Mult seems to stabilize reasonably well, and even so takes at least 1 billion references to do so. It appears that even to have the cache warm up one needs a trace of 500 million references!

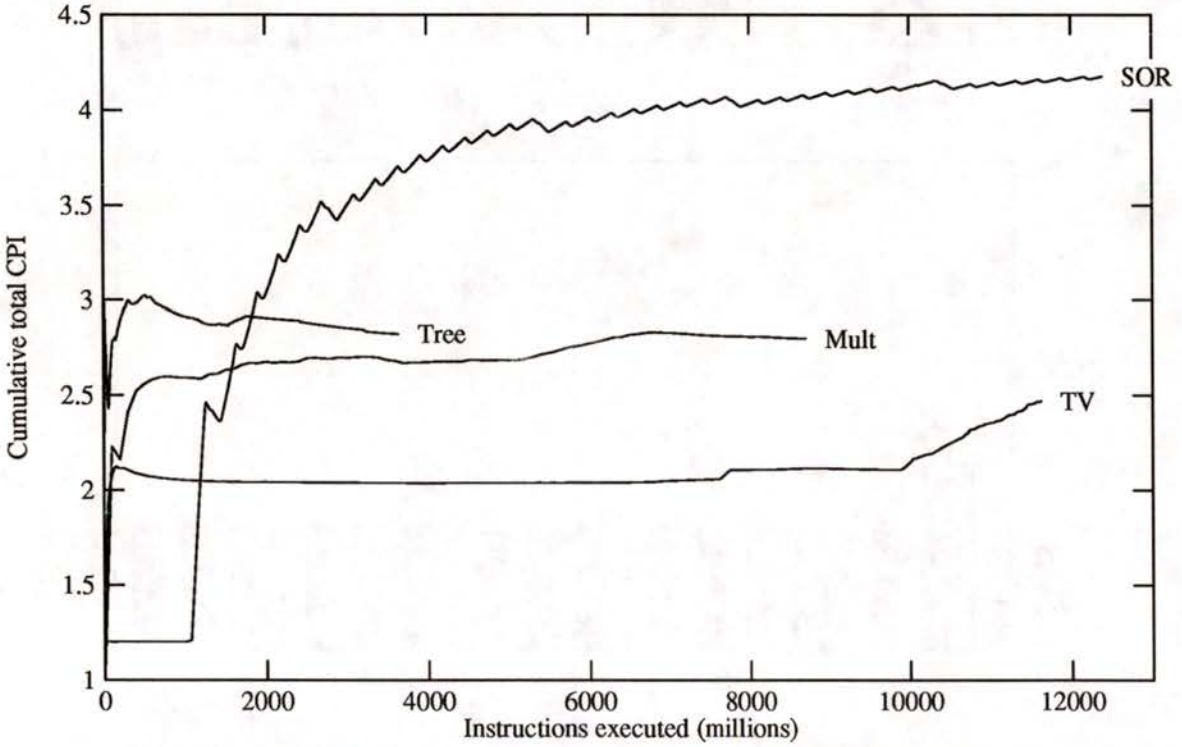


Figure 9: Cumulative CPI_{total} for the four examples with a 512K second level cache.

In each case, both first level caches had 4K bytes with 16 byte lines. The second level cache had 512K bytes in 128 byte lines. All were direct-mapped.

7.5. Address Mapping to the Second Level Cache

Recall that the trace addresses generated for user programs are virtual addresses, but the Titan and future machines at WRL will have physically addressed caches. Simulating small physically addressed caches using virtual addresses is not problematical. If the cache size divided by its associativity is less than or equal to the page size, then the high order bits of the page offset,

which is the same for virtual and physical addresses, can be used to identify the appropriate cache line. A process id field associated with each cache line determines whether the contents of the cache line is from the correct address space. In this case, misses will be identical to those occurring in a physical cache where the pid field is replaced by a physical page number field. When the cache is very much larger than the page size, as will almost certainly be the case for the second level cache, the page offset maps to only a tiny proportion of the cache.

If an address's offset relative to the cache size is used as a mapping, sequential virtual addresses will map sequentially into the cache, wrapping around only when they are larger than the cache size. Since the cache may be larger than the address space of many programs, all of those programs will conflict in those portions of the cache associated with commonly used virtual addresses. Even for large programs, instruction addresses, which map into the low addresses, will conflict.

A similar mapping based on physical addresses should provide better distribution throughout the cache. Physical addresses, which are sequential only within pages, will be scattered throughout the cache in page-sized chunks, assuming that the physical memory is very much larger than the second level cache. Their distribution in the cache will depend on the memory management algorithm. Unfortunately, using the actual physical address from the Titan in our simulations is unrealistic. Not only is the Titan's physical memory too small, but the physical addresses for user code correspond to the expanded trace code rather than the unexpanded real code.

Since the cache is so large, one should be able to spread out references from different processes, eliminating interprocess conflicts, and assuring that much of a process's working set remains in the cache after process switches. The two-level cache simulations performed by Panama use a simple pid hashing algorithm to distribute mappings in the cache. The goal of pid hashing is to reduce cache conflicts due to the "bunching" of references to particular areas in the address space. The pid is exclusive-ORed with the part of the address that indexes into the sets of the cache. Thus, the same virtual address from different address spaces will map into different areas of the cache, reducing conflicts. This method may be viewed as a possible implementation for a virtually addressed cache or as an approximation of the distribution naturally provided by physical addressing.

The graphs in Figure 10 compare the cumulative *CPI*s for three sizes of second level cache when pid hashing is used (dotted lines) and when the address modulo the cache size is used (solid lines). The upper graph shows CPI_{level2}. The lower graph shows CPI_{total}. Pid hashing effectively reduces CPI_{level2} only when the cache is quite large. Even in the best case, when it reduces the second level contribution for a 16 megabyte cache by about 30% it has less than 1% effect on the CPI_{total}. As cache size increases, pid hashing more effectively reduces the miss rate and thus the CPI_{level2}. On the other hand, as cache size increases, the miss rate and CPI_{level2} decrease so that even a large proportional reduction has a smaller effect on the CPI_{total}. A question to be considered is whether a heavier work load applied to a large cache would increase the second level miss ratio but still benefit greatly from pid hashing, or whether this would result in reduced effectiveness as in the case of the 512K cache.

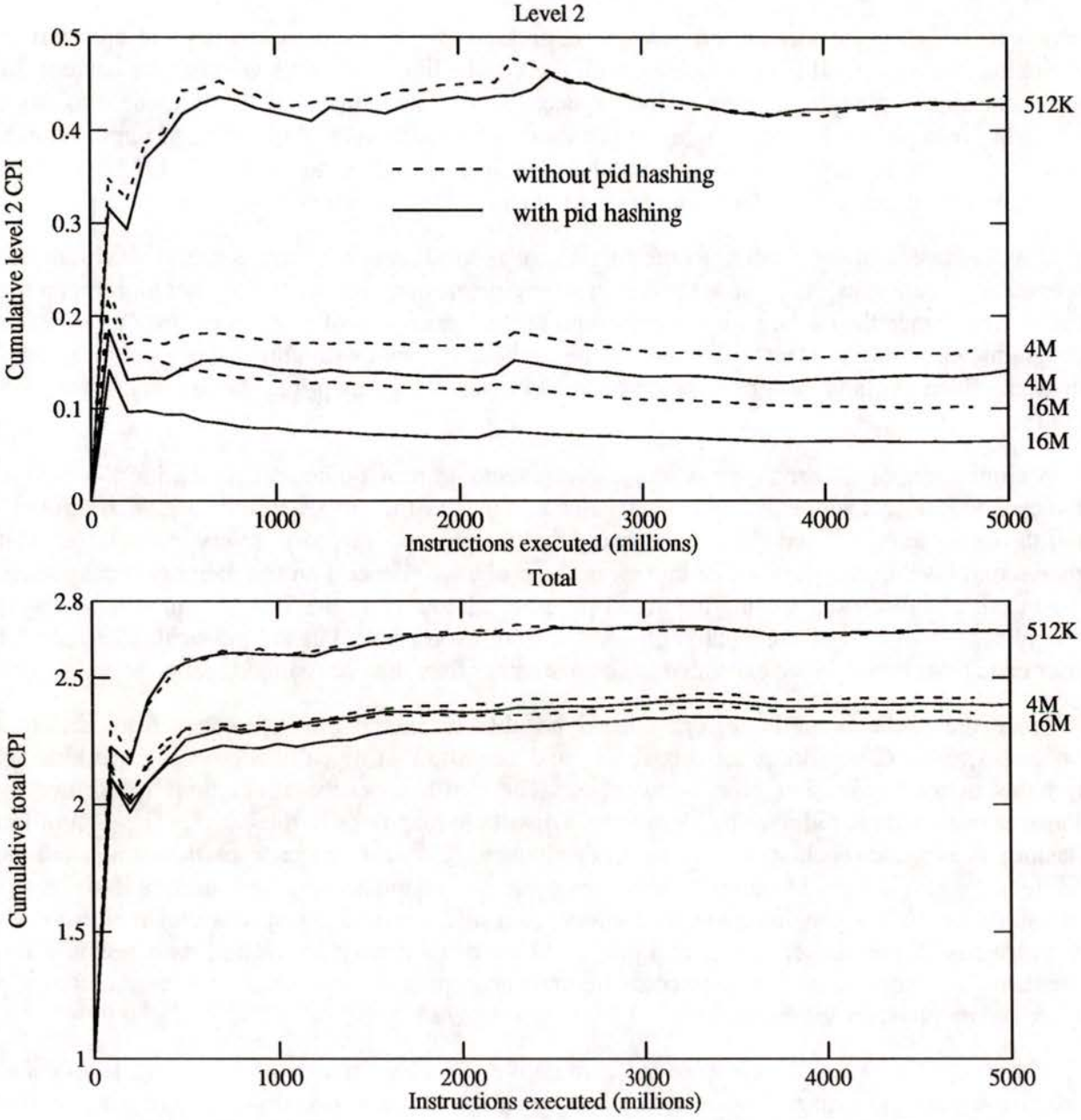


Figure 10: Effectiveness of pid hashing for 512K, 4M, and 16M second level caches

Cumulative *CPI*s for 5 billion instruction runs of Mult with pid hashing (solid lines) and without pid hashing (dotted lines). The upper graph shows only the CPI_{level2}. The lower graph shows the CPI_{total}.

7.6. Multiple Process Traces

A feature of our system is that we do not need to save traces. To analyze a program with a different cache organization, we need only rerun the program with a differently parameterized analysis program. This works well for single user programs that are deterministic; they generate identical traces on different runs. For multiple processes, however, the pattern of memory references is no longer deterministic, resulting in variations in cache behavior from run to run. The same set of programs can be interleaved differently depending on a variety of external uncontrollable factors. This is not necessarily bad. Differences in the results for the same set of

programs represent differences that will occur on a real machine. On the other hand, variations between runs may make it difficult to precisely understand the significance of variations due to cache modifications.

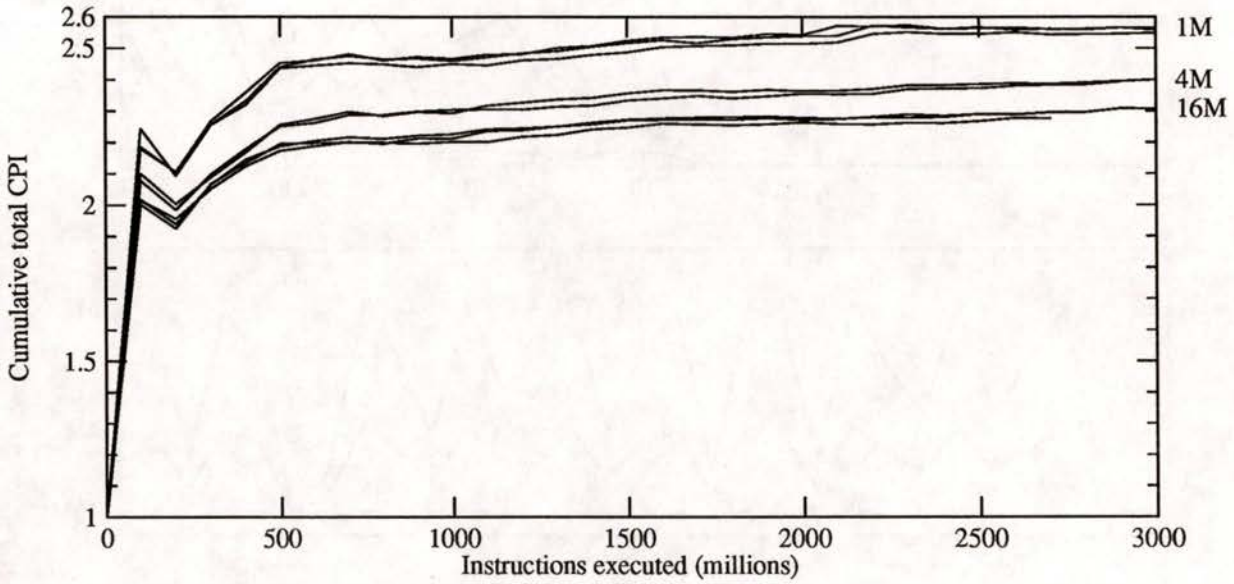


Figure 11: Variations in multiprocess runs

For all eight runs the first level caches were 4K bytes with 16 byte lines. The second level caches were 1M and 4M bytes with 128 byte lines and 16M with 256 byte lines.

Figure 11 shows the cumulative CPI_{total} for repeated runs of Mult at three second level cache sizes. The variations between same sized runs are quite small, usually less than 1%. In all of our experiments, variations in cache characteristics caused changes in the *CPI* results that were either significantly and consistently greater than 1% or the fact that they were smaller was by itself a useful result. In cases were more specific detailed distinctions between similar results are needed, we will modify the cache analysis program to simulate multiple caches at the same time. The need for this has not yet risen.

7.7. Second Level Cache Size

Next, we present the results of varying only the size of the second level cache. We did not simulate the case without a second level cache but can easily argue that its performance would be unacceptable. Assuming miss ratios of 5% for the data and instruction caches, a 200 cycle cost for each miss, and that the data cache is used 40% as often as the instruction cache, the CPI_{total} would be 15! A slower processor in which the time to handle a miss was 10, would have a CPI_{total} of 1.7. The actual increase in performance of the machine with a 20 fold increase in processing speed, would be only 2 times.

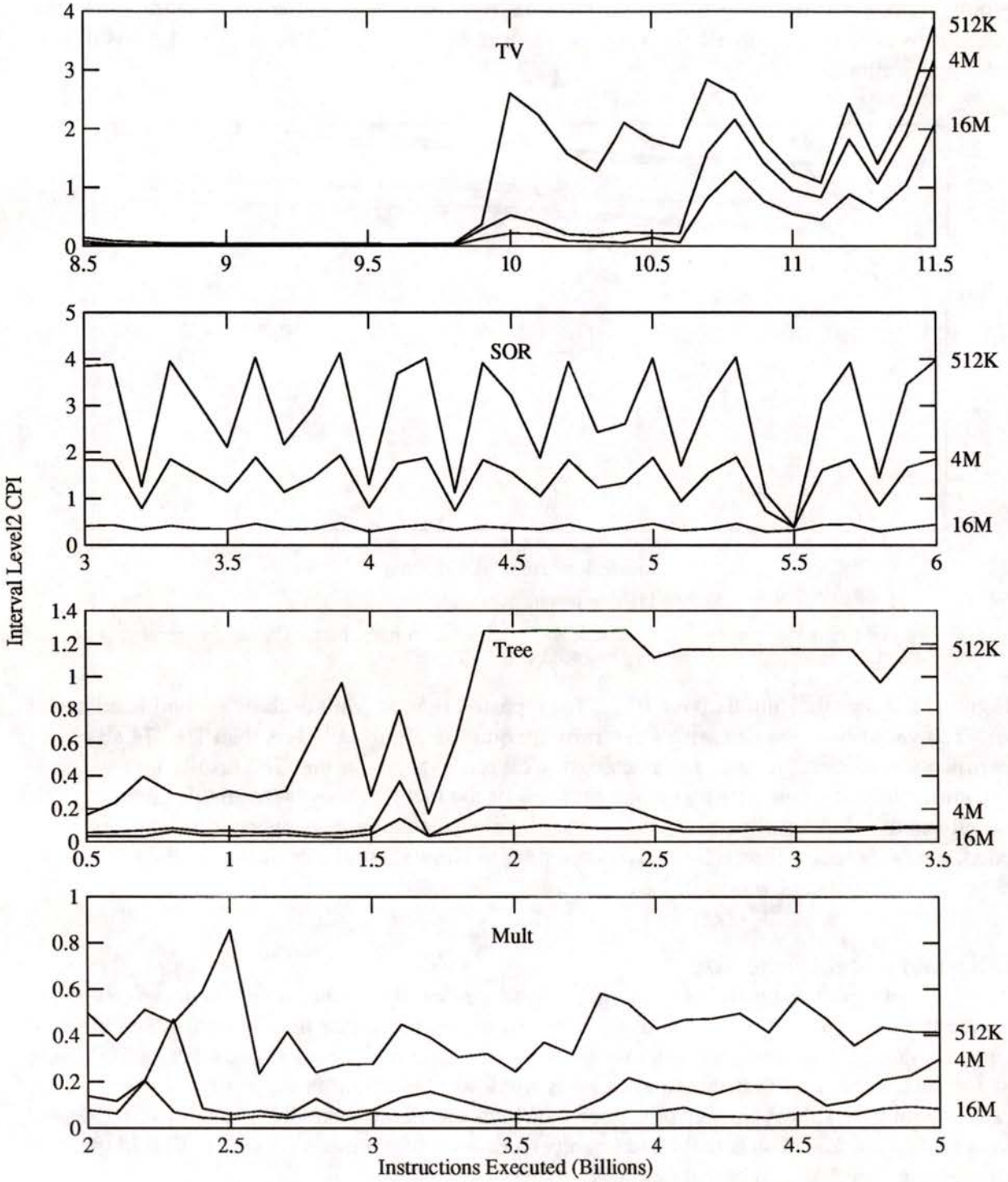


Figure 12: Interval CPI_{level2} for three second level cache sizes

CPI_{level2} for the previous 100M instructions plotted against instructions executed for three second level cache sizes: 512K, 4M, and 16M. Note the difference in the scale on the y-axis and the difference in trace length on the x-axis.

Figure 12 shows the CPI_{level2} for three sizes of the second level cache: 512K, 4M, and 16M. Except during the early part of the TV run, there is a dramatic reduction in CPI_{level2} when the cache size is increased to 4M but a considerably smaller improvement resulting from the jump to 16M.

Interval *CPI* gives a good picture of the different behaviors of the programs, however, the cumulative *CPI*s shown in Figure 13 shows the effect of cache size increases on long run average system performance. SOR, with its periodic behavior, is the only program that greatly benefits from an increase to 16M. This is most likely because it actually references a large part of its large address space. We can conclude from these examples that most of the programs we have chosen do not have large working sets. Mult is the most realistic of the examples, however, its usefulness is limited by the relatively small size of each of the individual programs it runs. There is definitely a need to put together much larger multiprocessing workloads.

We can get a bit more insight into the existing data by dividing level 2 cache misses into three categories defined by Hill [6]: *compulsory*, *capacity*, and *conflict*. Compulsory misses happen the first time an address is referenced and and occur regardless of the size of the cache. Our simulator computes the compulsory, fully associative, and direct-mapped miss ratios. The capacity miss ratio is the difference between the fully associative miss ratio and the compulsory miss ratio. The conflict miss ratio is the difference between the direct-mapped miss ratio and the associative miss ratio. From these values we have computed the percent of misses in each category for the Mult runs for each cache size shown in Figure 14.

Capacity misses are due to fixed cache size. The percent of capacity misses decreases markedly when the cache size in increased to 4M bytes, but very little more as the result of the increase to 16M. Conflict misses result from too many mappings of addresses to the same cache lines. The percent of conflict misses increases slightly from 512K to 4M, but decreases substantially when the cache size is increased to 16M. Thus, beyond 4M, increases in size primarily allow interfering blocks to be spread out more effectively. This corresponds to the increased effectiveness of pid hashing to reduce the miss rate for a 16M cache.

7.8. Direct-Mapped vs Associative Second Level Caches

In this section, we examine the effects of associativity in large second level caches. Increasing associativity can decrease the miss ratio of a cache, but it may have limited effect or even an adverse effect on the performance of the system as a whole [5]. If the cache is very large, having a low miss ratio in the direct-mapped case, then reducing the miss ratio can only have a small effect on overall performance, even if associativity comes with no cost. It is more likely that for large caches, the cost of implementing associativity outweighs its benefits. The positive effects of associativity are related to the cache's miss ratio, but the negative effects in added cost are related to the total number of references to the cache. Though increased associativity may decrease the number of misses and thus the total cost of misses, it makes every reference to a cache more expensive.

Our simulations measure the miss ratios and *CPI*s for direct-mapped and fully associative caches under the extremely optimistic assumption that associativity is free. That is, the cost of a reference in both cases is assumed to be equal. LRU replacement is used in the fully associative case. The results should bound the improvement possible from set associativity. Figure 15

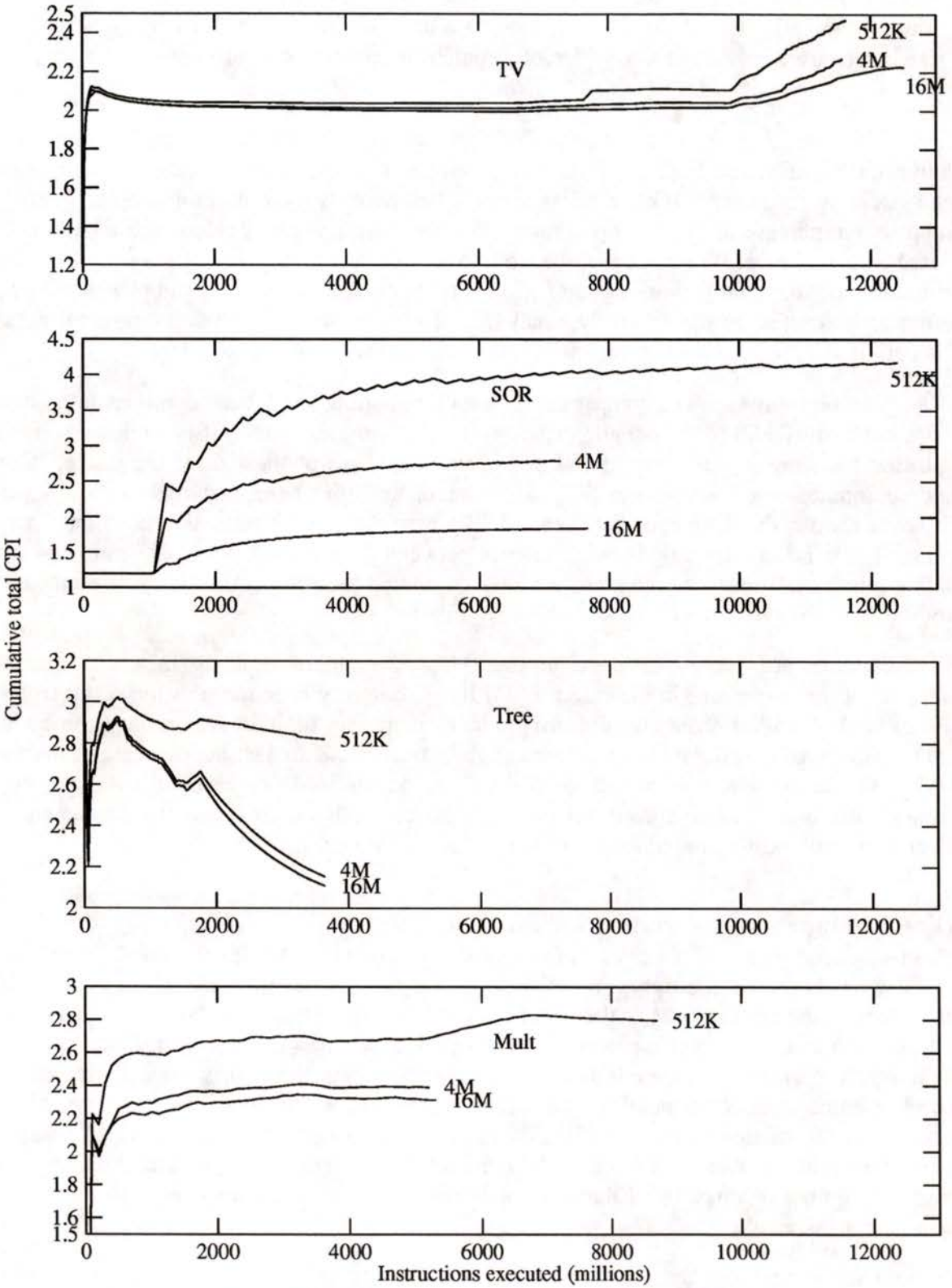


Figure 13: Cumulative CPI_{total} for three second level cache sizes

Cumulative CPI_{total} for the four example programs with three second level cache sizes: 512K, 4M, and 16M. Only SOR substantially benefits from an increase in size to 16M. Note the differences in scale on the y-axis.

Categories of level 2 misses			
Cache Size	conflict (%)	capacity (%)	compulsory (%)
512K	40.4	57.6	2.0
4M	41.5	47.3	11.2
16M	32.0	45.1	22.9

Figure 14: Compulsory, capacity, and conflict misses

shows the cumulative CPI_{total} in the direct-mapped and associative cases for Mult runs with four second level cache sizes. The results show that even in the most optimistic case, associativity is minimally effective for large caches (4M and 16M) but may be useful for 1M or 512K caches.

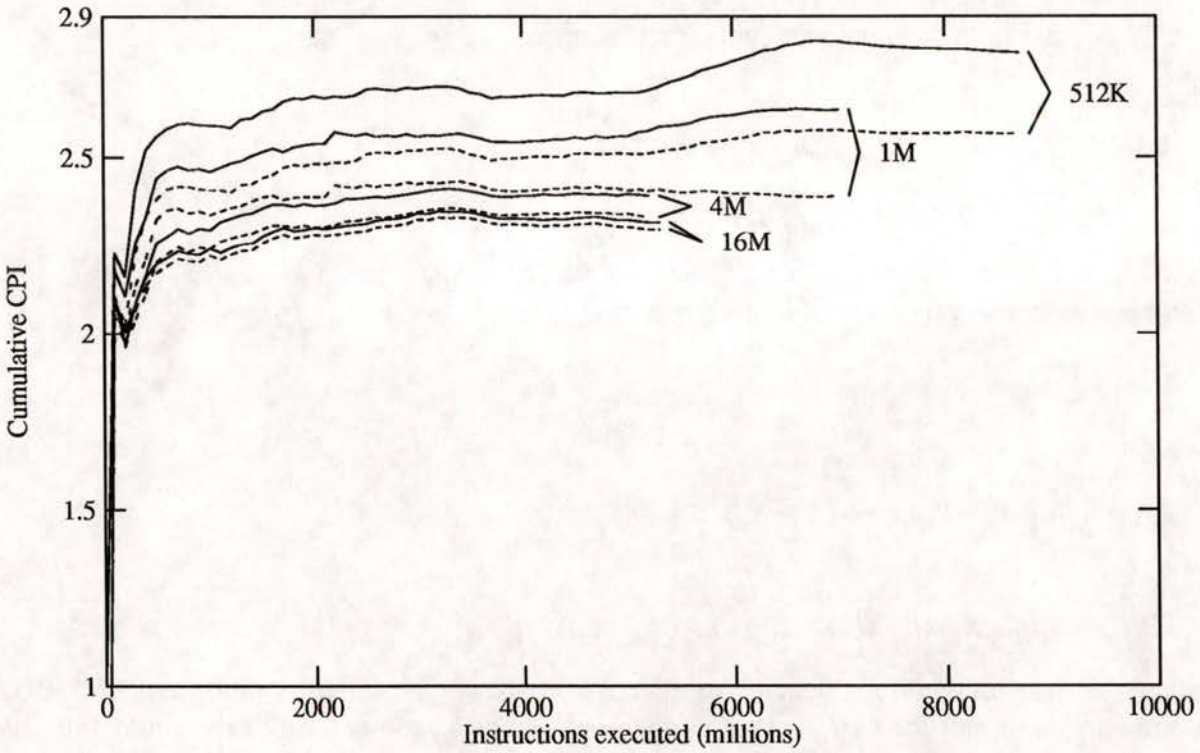


Figure 15: CPI_{total} for direct mapped and fully associative cases

Cumulative CPI_{total} for Mult for direct-mapped (solid) and fully associative (dotted) for four cache sizes.

A more useful measure to implementors of the benefits of associativity is the effect on the total *CPI* of associativity as a function of the cost per reference of the associativity. We derive the formulas for this below.

Using subscripts a and d to indicate associative and direct-mapped, respectively,

$$CPI_a = CPI_d + \Delta CPI$$
$$= CPI_d + R_1 \cdot (A_a - A_d)$$

where A_a and A_d are the average cost per reference to the second level cache, and R_1 is the first level miss ratio (combined for data and instruction caches).

Letting

$$\Delta A = A_a - A_d$$

and

h_a = cost of a hit in the associative cache

h_d = cost of a hit in the direct-mapped cache

m = cost of a second level cache miss (same direct-mapped and associative)

R_a = miss ratio for a second level associative cache

R_d = miss ratio for a second level direct-mapped

then,

$$\Delta A = h_a \cdot R_a + m \cdot (1 - R_a) - (h_d \cdot R_d + m \cdot (1 - R_d))$$
$$= h_a \cdot R_a - h_d \cdot R_d + m \cdot (R_d - R_a)$$

Expressing this as a function of h_d and the ratio

$$k = \frac{h_d}{h_a}$$

we get

$$\Delta A = h_d \cdot (\frac{1}{k} \cdot R_a - R_d) + m \cdot (R_d - R_a)$$

and

$$CPI_a = CPI_d + R_1 \cdot (h_d \cdot (\frac{1}{k} R_a - R_d) + m \cdot (R_d - R_a))$$

Since the miss ratios do not depend on reference costs, we use the miss ratios and CPI_d from our simulations to plot the CPI_a versus k for the values of h_d and m used in the simulation. he results are shown in Figure 16. The horizontal lines represent CPI_d, the direct-mapped value, for each cache size. The corresponding sloping lines represent CPI_a, the fully associative value, for various values of k. The diamonds note the crossover points at which the CPI is the same in the associative and direct-mapped cases. The distance between the points at which corresponding horizontal and sloping lines intersect the vertical at $k = 1$ is the maximum gain to be expected from associativity, assuming that full associativity provides an upper bound on that gain.

The graphs for Mult, Tree and TV show that associativity will not help for a second level cache of 16M or probably even 4M. For caches sizes of 512K or less, we should do more

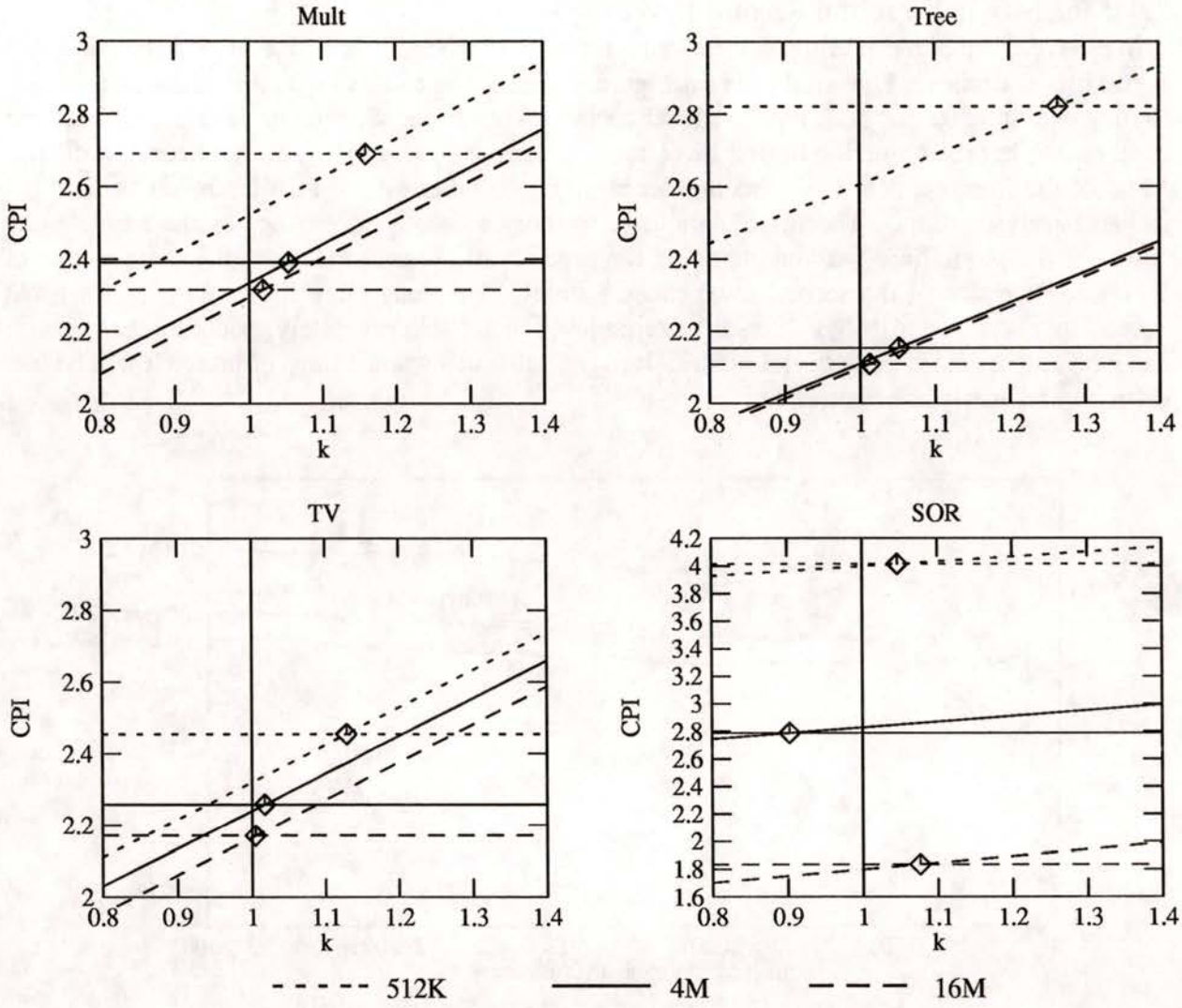


Figure 16: *CPI*s for fully associative caches as function of implementation cost

detailed simulations using realistic costs and degrees of associativity. In these three case, as expected, all diamonds are to the right of $k = 1$. That is, performance improves with full associativity. Yet, the intersection in the 16M case is very close to $k = 1$ implying that associativity would have to be nearly free to get even a small gain. On the other hand, for a cache size of 512K, if associativity could be implemented for a cost per reference of 10% or less, all three of the benchmarks would see some improvement in performance.

SOR is anomalous because in the 4M case, full associativity does worse than direct-mapped. This is a result of the size of data structures and the cyclic way they are used. Evidently, the LRU replacement strategy results in the frequent replacement of data items which are about to be used. Clearly, increasing cache size has a much greater effect on SOR than does increasing associativity.

7.9. Line Size in First and Second Level Caches

We have done some preliminary experiments with changes in line size in both the first and second level caches. The solid lines in Figure 17 show the effects of doubling and then quadrupling the line size for both the first level caches. These runs were made with a 512K second level cache, but the reduction in first level contributions are independent of the second level size. Most of the improvement is due to the deceased contribution of the instruction cache. CPI_{data} stayed nearly constant. The improvement in memory system performance as the result going from 16 to 32 byte lines without changing the capacity of the cache is about the same as that of increasing the size of the second level cache 8 times. The dotted line shows CPI_{total} for a 4M second level cache and 16byte lines in the first level cache. Unfortunately, doubling the line size of an on chip cache is not a trivial matter. It may be that using some form of prefetch will be less difficult and nearly as effective.

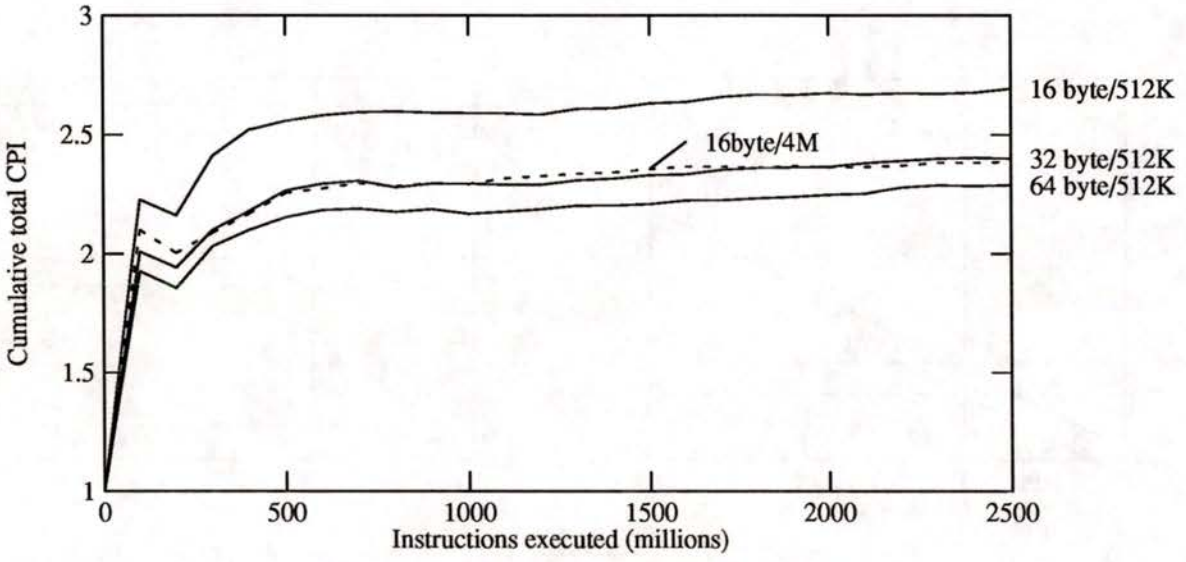


Figure 17: Three lines sizes for the first level caches

Solid lines show variations in the line size of the first level caches with a second level cache size of 512K. The dotted line is from a run with a 4M second level cache and 16 byte lines in the first level cache.

Doubling the length of lines in the second level cache from 128 bytes to 256 bytes made almost no difference in performance as shown in Figure 18. The difference is within the 1% variation found between runs of Mult with constant cache characteristics. This is the case even though the cost of retrieving the two line sizes was identical in the tow runs. Apparently, there is too much conflict for long line sizes to be beneficial.

7.10. Write Buffer Contributions

We were initially surprised to find that there was a great deal of variation in the performance of the write buffer from program to program. In SOR, the $CPI_{wtbuffer}$ was nearly alway zero, but in runs of Mult and Tree it was as high as .5 cycles, or nearly 20% of the total CPI over a 100 million instruction interval.

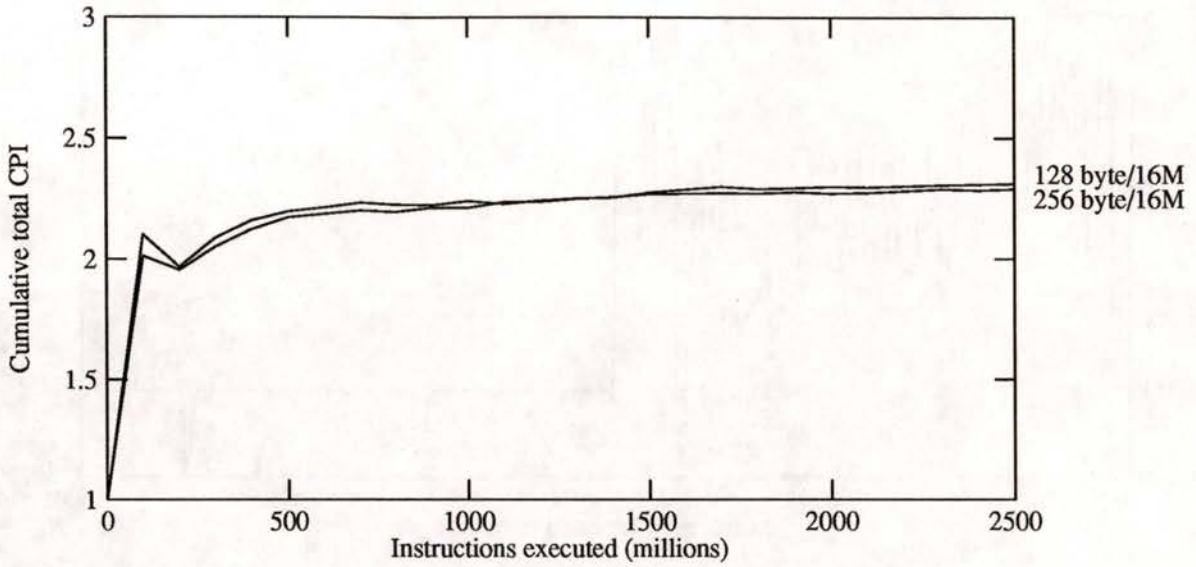


Figure 18: Two line sizes a 16M second level cache

Closer examination showed the reason to be the the sensitivity of write buffer performance to both the proportion and distribution of writes in the instructions executed. A write buffer entry is retired every six cycles. One would expect it to perform badly if on average more than one in six instructions was a write. In addition, bunching of the writes, rather than even distribution compounds the problem.

The graphs in Figures 19 and 20 show the clear relationship between the proportion of writes in a program and the write buffer performance. Whenever the percent of writes goes about one in six or 17%, the $CPI_{wtbuffer}$ takes a proportionally greater jump. The change in $CPI_{wtbuffer}$ in the 2000M-5000M instruction range in the Mult graph of Figure 19 is most likely the result of the smaller run of Tree executed as part of the Mult set of programs. This lead us to look more closely at the difference between the compiled Scheme code for Tree and that for the other programs.

To determine this correlation more precisely, we plotted the $CPI_{wtbuffer}$ against percent of instructions that are writes. The graphs in Figures 21 and 22 show both the effect of the proportion of writes and the distribution of writes. Although our simulations do not measure the distribution of writes, some trends are apparent from the graphs. The tendency of $CPI_{wtbuffer}$ values associated with a percent of writes to be very close together indicates a consistent degree of bunching. Variations in the distribution of writes shows up as a spread of values for each percentage.

Inspection of the code generated for Tree showed that nearly all of the store instructions are used to save values in registers on procedure entry. Because of the large number of very small procedures and the recursive nature of the program, it is nearly impossible for global register allocation to optimize out the stores. Thus, there are often both a high number of stores and the stores are necessarily bunched together. The bunching was not an artifact of trace generation. As a result, on most procedure entries, the write buffer becomes full causing delays and a positive contribution to the *CPI*.

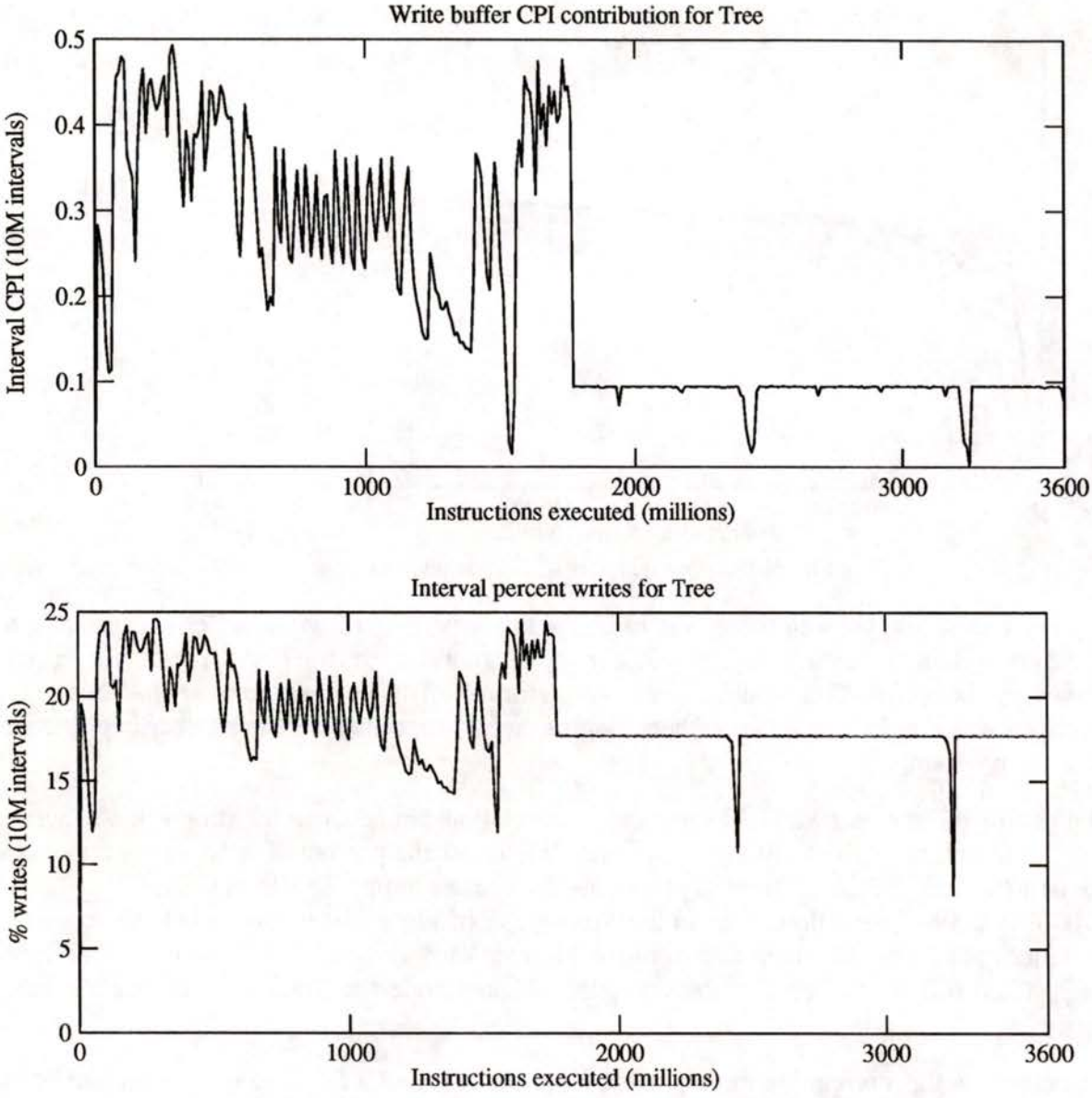


Figure 19: $CPI_{wtbuffer}$ compared with proportion of writes in Tree.
All runs are identical so only one is shown.

The graph for Mult shows greater variation. The grouping of points above 0.2 *CPI* corresponds to the influence of the included Tree run averaged with other processes with which it interleaves execution. Below 15% writes, the *CPI* is mostly dependent on the degree of bunching. Not surprisingly, this varies most in the multiprocess example. We would expect to see more bunching of writes and worse write buffer performance had we used per-procedure rather than global register allocation. In the absence of recursion, global register allocation very effectively reduces the number of loads and stores necessary on procedure call and return [17, 19].

For TV, the lower $CPI_{wtbuffer}$ for higher percentages of writes shows that writes are more evenly distributed. This is expected because TV does not do nearly as many procedure calls as either Mult or Tree.

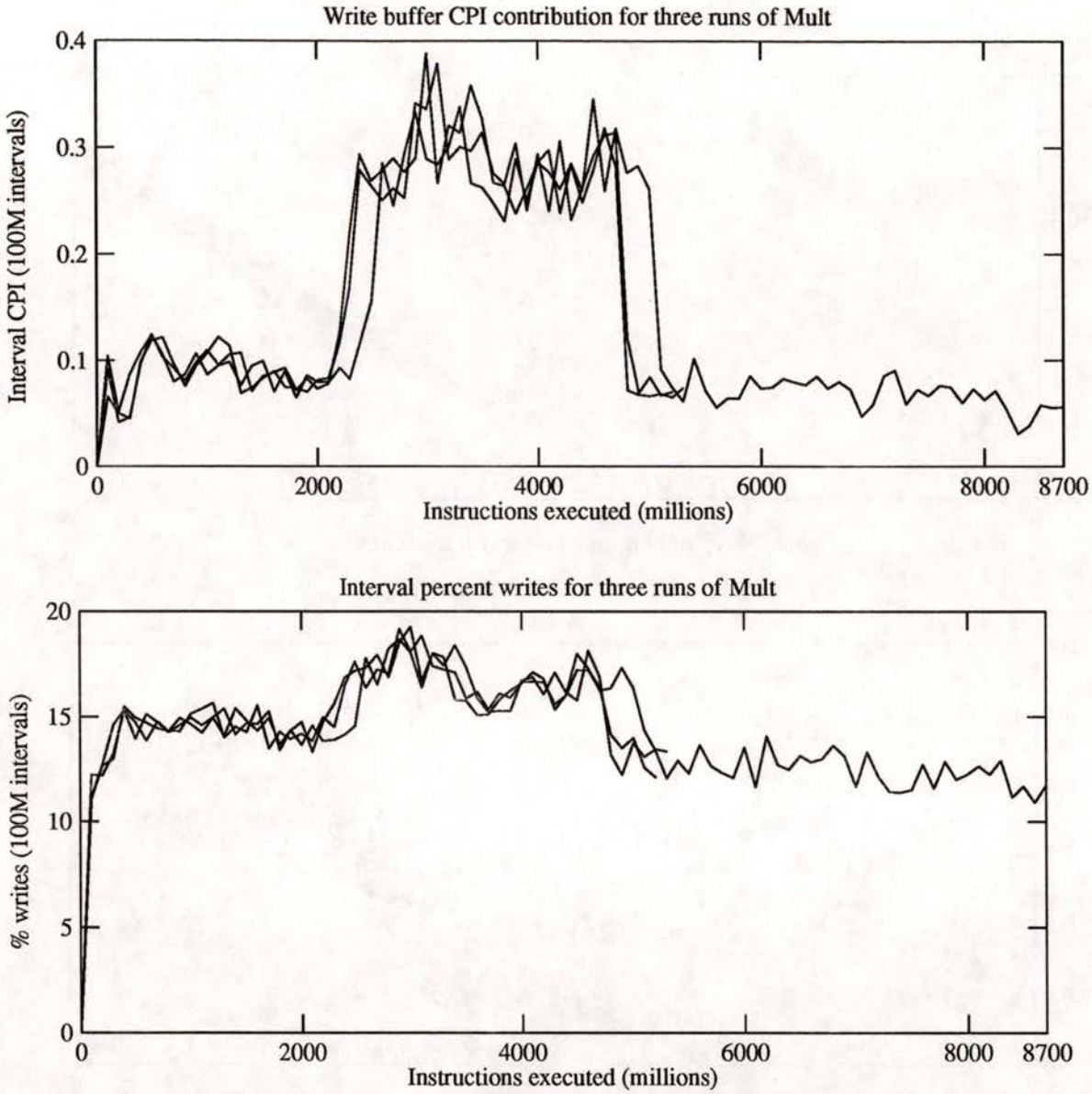


Figure 20: $CPI_{wtbuffer}$ compared with proportion of writes in Mult. Three runs are shown.

We believe that a third factor, the first level data cache miss ratio, comes into play for SOR. There, even though the percent of writes is frequently in the 15-20% range, $CPI_{wtbuffer}$ is usually zero. If writes are uniformly distributed below 18% the write buffer will never fill and a subsequent write will never be delayed. At points above 18% we expect some delays. Since the second level cache is assumed to be pipelined into 3 stages, the processing of a single miss gives the write buffer a chance to retire two entires. Frequent enough misses interspersed among the writes could cause the write buffer to work smoothly without delays. We do not yet have the data to verify this hypothesis.

In programs with large basic blocks our method of generating traces can exaggerate the write buffer's contribution to the *CPI*. Since there is only one instruction entry per basic block in a

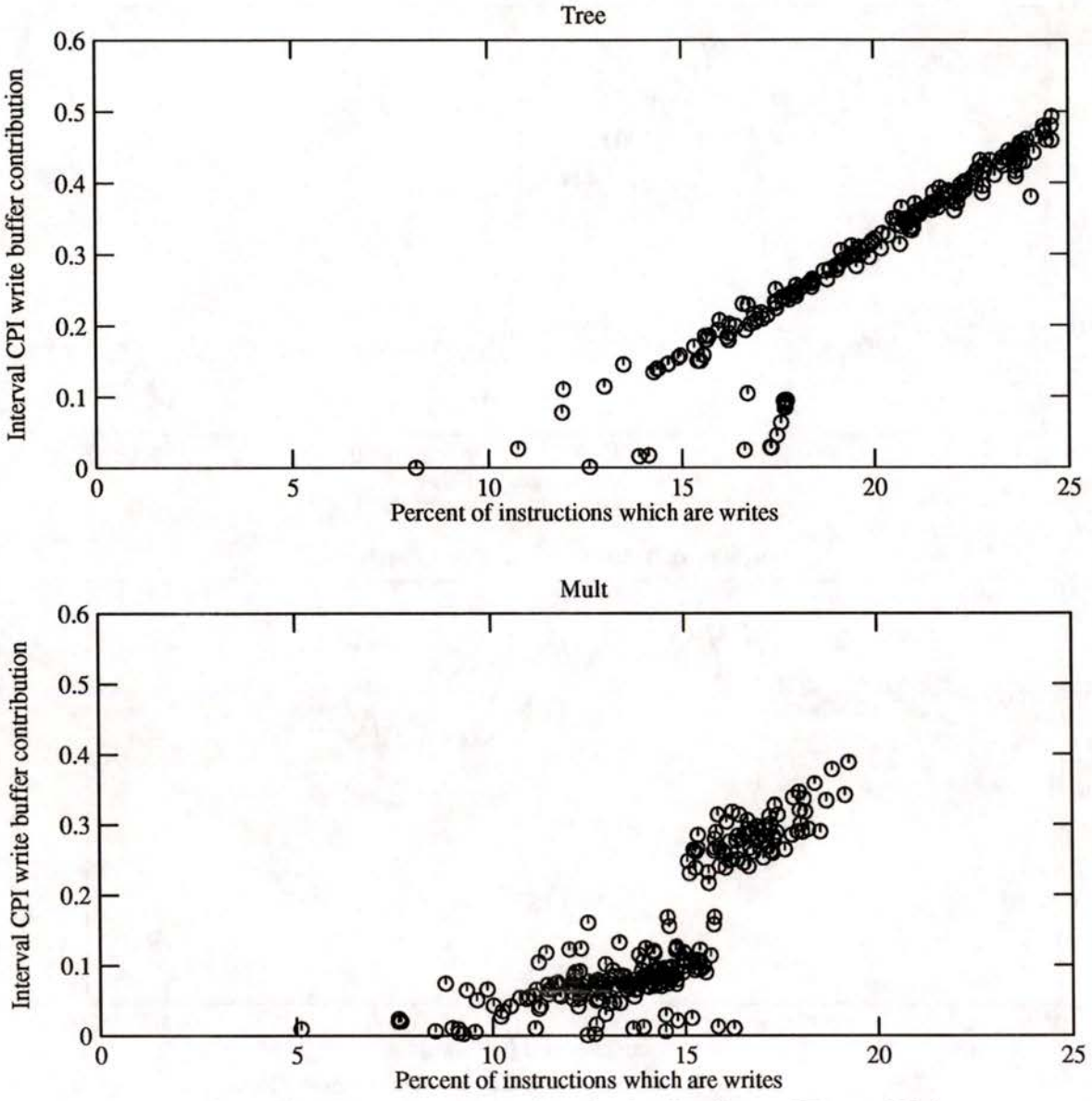


Figure 21: $CPI_{wtbuffer}$ vs proportion of writes for all runs of Tree and Mult

trace, we simulate all instruction references followed by all data references in their correct order. Writes that are interspersed with other instructions will be bunched together more in the traces than in reality. The larger the basic block, the more likely that writes at the end will be sufficiently bunched to show up as write buffer delays when simulated. We expect that simulations of numeric programs which have been optimized using loop unrolling, resulting in greatly increased basic block size, would suffer from this distortion.

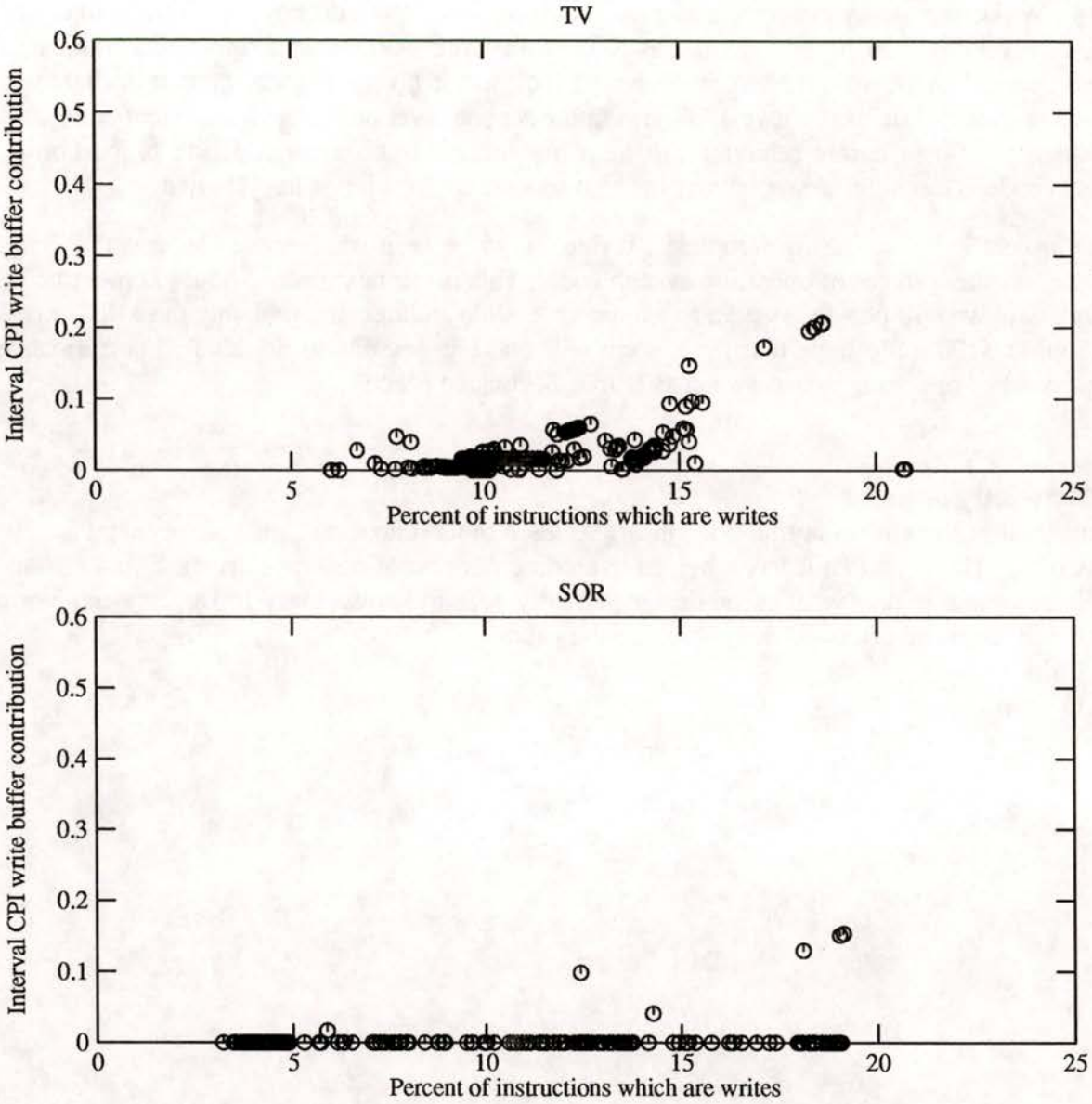


Figure 22: $CPI_{wtbuffer}$ vs proportion of writes for all runs of TV and SOR

8. Conclusion

We have presented a new software method for generating extremely long traces of program execution on RISC machines. The system is designed to allow tracing of multiple user processes in different address spaces and the operating system. On-the-fly analysis solves the storage problem for the billions of bytes generated. On-the-fly data collection allows long traces to be compacted and written to tape if necessary. The system is easy to use, requiring only the relinking of programs to be traced.

We have used the system to generate and analyze traces for a variety of single and multiuser programs. A conservative estimate of the number of references we have traced and analyzed is 500 billion. We have produced tapes of compressed traces containing 45 billion references.

The trace analysis has provided insights into the behavior of systems with large second level caches. While secondary caches are clearly necessary, our traces do not justify sizes over 4M bytes. Only individual programs such as SOR with large working sets rather than just large address spaces benefit from the largest caches. Block size is a very important factor in first level instruction caches, but sizes above 128 bytes in the second level cache had little effect on overall performance. Write buffer behavior can be a big factor in performance if the proportion of writes in code is too high. Associativity in large second level caches is not justified.

The project is by no means complete. It remains to be seen whether we can generate long, accurate, seamless traces of operating system code. This is our next goal. Should kernel tracing be successful,we will port the system to a more accessible architecture, probably the MIPS-based DECStation 3100. We hope that the system will make it possible to do detailed performance comparisons of operating systems such as Ultrix, Sprite and Mach.

Acknowledgements

Mark Hill at the University of Wisconsin provided the Tycho cache analysis package for our use. Colleen Hawk and Paul Vixie helped overcome our fear of new tape drives. Alan Eustace and Norm Jouppi told us what cache designers really need to know. Mary Jo Doherty and Norm Jouppi corrected and critiqued early drafts of the paper.

References

[1] Agarwal, A., Sites, R., and Horowitz, M.
ATUM: A New Technique for Capturing Address Traces Using Microcode.
In *13th Annual Symposium on Computer Architecture*, pages 119-127. June, 1986.

[2] Bartlett, J. F.
SCHEME->C: A Portable Scheme-to-C Compiler.
WRL Research Report 89/1, Digital Equipment Western Research Laboratory, 1989.

[3] De Leone, R. and Mangasarian, O. L.
Serial and Parallel Solution of Large Scale Linear Programs by Augmented Lagrangian Successive Overrrelaxation.
In Kurzhanski, A., Neuman, K., and Pallaschke, D. (editors), *Lecture Notes in Economics and Mathematical Systems 304: Optimization, Parallel Processing and Applications*, pages 103-124. , 1988.

[4] Dion, J.
Fast Printed Circuit Board Routing.
WRL Research Report 88/1, Digital Equipment Western Research Laboratory, 1988.

[5] Hill, M. D.
A Case for Direct-Mapped Caches.
IEEE Computer 21(12):25=40, December, 1988.

[6] Hill, M., and Smith, A.
Evaluating Associativity in CPU Caches.
Computer Science Technical Report 823, University of Wisconsin, February, 1989.
To appear in *IEEE Transactions on Computers* C-38(12),December,1989.

[7] Jouppi, N. P.
Timing Analysis and Performance Improvement of MOS VLSI Designs.
IEEE Transactions on Computer Aided Design 6(4):650-665, 1987.

[8] Mangasarian, O. L.
Sparsity-Preserving SOR Algorithm for Separable Quadratic and Linear Programming.
Comput. and Ops. Res. 11(2), 1982.

[9] Mattson, R., Gecsei, J., Slutz, R., and Traiger, I.
Evaluation Techniques for Storage Hierarchies.
IBM Systems Journal 9(2), September, 1970.

[10] McFarling, S.
Program Optimization for Instruction Caches.
In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183-191. April, 1989.

[11] Ousterhout, J., Hamachi, G., Mayo, R., Scott, W., and Taylor, G.S.
The Magic VLSI Layout System.
IEEE Design and Test of Computers 2(1):19-30, February, 1985.

[12] Samples, D.
Mache: No-Loss Trace Compaction.
In *International Conference on Measurement and Modeling of Computer Systems.* , 1989.

[13] Sites, R.
Personal Communication.
1988.

[14] Smith, A. J.
Cache Memories.
ACM Computer Surveys 14(3):473-530, September, 1982.

[15] Stark, D. and Horowitz, M. .
Techniques for Calculating Currents and Voltages in VLSI Power Supply Networks.
To appear in *IEEE Transactions on Computer Aided Design* 2, february, 1990.

[16] Stunkel, C., and Fuchs, W.
TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation.
In *International Conference on Measurement and Modeling of Computer Systems.* , 1989.

[17] Wall, D. W. .
Global Register Allocation at Link-time.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 264-275. 1986.
Also available as WRL Technical Report 86/3.

[18] Wall, D. W., and Powell, M. L.
The Mahler Experience: Using an Intermediate Language as the Machine Description.
In *Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 100-104. 1987.
A more detailed version is available as WRL Technical Report 87/1.

[19] Wall, D. W.
Register Windows vs. Register Allocation.
WRL Research Report 87/5, Digital Equipment Western Research Laboratory, 1987.

[20] Ziv, J. and Lempel, A.
A Universal Algorithm for Sequential Data Compression.
IEEE Transactions on Information Theory (23):75-81, 1986.

Unix is a registered trademark of AT&T.
Ultrix is a trademark of Digital Equipment Corporation.
Sony is a registered trademark of Sony Corporation.
Exabyte is a trademark of Exabyte Corporation.

WRL Research Reports

"Titan System Manual."
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

"Global Register Allocation at Link Time."
David W. Wall.
WRL Research Report 86/3, October 1986.

"Optimal Finned Heat Sinks."
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

"The Mahler Experience: Using an Intermediate Language as the Machine Description."
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

"The Packet Filter: An Efficient Mechanism for User-level Network Code."
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

"Fragmentation Considered Harmful."
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

"Cache Coherence in Distributed Systems."
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

"Register Windows vs. Register Allocation."
David W. Wall.
WRL Research Report 87/5, December 1987.

"Editing Graphical Objects Using Procedural Representations."
Paul J. Asente.
WRL Research Report 87/6, November 1987.

"The USENET Cookbook: an Experiment in Electronic Publication."
Brian K. Reid.
WRL Research Report 87/7, December 1987.

"MultiTitan: Four Architecture Papers."
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

"Fast Printed Circuit Board Routing."
Jeremy Dion.
WRL Research Report 88/1, March 1988.

"Compacting Garbage Collection with Ambiguous Roots."
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

"The Experimental Literature of The Internet: An Annotated Bibliography."
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

"Measured Capacity of an Ethernet: Myths and Reality."
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

"Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description."
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

"SCHEME->C A Portable Scheme-to-C Compiler."
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

"Optimal Group Distribution in Carry-Skip Adders."
Silvio Turrini.
WRL Research Report 89/2, February 1989.

"Precise Robotic Paste Dot Dispensing."
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

"Simple and Flexible Datagram Access Controls for Unix-based Gateways."
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

"Spritely NFS: Implementation and Performance of Cache-Consistency Protocols."
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

"Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines."
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

"A Unified Vector/Scalar Floating-Point Architecture."
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

"Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU."
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

"Integration and Packaging Plateaus of Processor Performance."
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

"A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance."
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

"Leaf: A Netlist to Layout Converter for ECL Gates."
Robert L. Alverson and Norman P. Jouppi.
WRL Research Report 89/12, July 1989.

"The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance."
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

"Long Address Traces from RISC Machines: Generation and Analysis."
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

"Link-Time Code Modification."
David W. Wall.
WRL Research Report 89/17, September 1989.

WRL Technical Notes

"TCP/IP PrintServer: Print Server Protocol."
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

"TCP/IP PrintServer: Server Architecture and Implementation."
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

"Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer."
Joel McCormack.
WRL Technical Note TN-9, September 1989.

Table of Contents

1. Introduction 1
   1.1. Background 1
   1.2. Goals and Status 2
2. Trace Generation 3
   2.1. Environment: The Titan 3
   2.2. Link-time Code Modification 3
   2.3. Operating System Support 5
   2.4. The Trace Code 6
   2.5. Controlling Tracing 6
   2.6. Explicit Trace Buffer Entries 7
   2.7. Format of Trace Buffer Entries 7
   2.8. Kernel Tracing: A Few Special Considerations 8
3. Trace Management 8
   3.1. Extracting the Traces from Memory 8
   3.2. Reproducibility of Results 9
4. Accuracy of the Trace Data 9
   4.1. Eliminating Gaps for Seamless Traces 10
   4.2. Trace-Induced TLB Faults 10
   4.3. Interrupt Timing 11
   4.4. Process Switch Interval 11
5. Verification 12
   5.1. Comparison with Simulation Results 12
   5.2. Checking for Gaps 12
6. Trace Analysis 13
   6.1. Panama: A Cache Analysis Program 13
   6.2. Tycho: Evaluating Associativity 14
   6.3. Saving Traces for Later Analysis 14
7. Experiments with Single and Multiple Process User Traces 14
   7.1. Base Case Cache Organization Assumptions 14
   7.2. Choice of User Programs 15
   7.3. *CPI* as a Measure of Cache Behavior 17
   7.4. Long Traces are Nice and Necessary 18
   7.5. Address Mapping to the Second Level Cache 22
   7.6. Multiple Process Traces 24
   7.7. Second Level Cache Size 25
   7.8. Direct-Mapped vs Associative Second Level Caches 27
   7.9. Line Size in First and Second Level Caches 32
   7.10. Write Buffer Contributions 32
8. Conclusion 37
Acknowledgements 38
References 39

List of Figures

Figure 1: Mapping of the shared trace buffer 5
Figure 2: Data entry format 7
Figure 3: Instruction entry format 8
Figure 4: Descriptions of programs used in trace experiments 16
Figure 5: User trace characteristics 17
Figure 6: Interval miss ratios (top) and interval *CPI*s (bottom) for Mult 19
Figure 7: Interval *CPI*s for TV in two parts 20
Figure 8: Miss ratios for short (500K refs) and long (150M refs) traces 21
Figure 9: Cumulative CPI_{total} for the four examples with a 512K second level cache. 22
Figure 10: Effectiveness of pid hashing for 512K, 4M, and 16M second level caches 24
Figure 11: Variations in multiprocess runs 25
Figure 12: Interval CPI_{level2} for three second level cache sizes 26
Figure 13: Cumulative CPI_{total} for three second level cache sizes 28
Figure 14: Compulsory, capacity, and conflict misses 29
Figure 15: CPI_{total} for direct mapped and fully associative cases 29
Figure 16: *CPI*s for fully associative caches as function of implementation cost 31
Figure 17: Three lines sizes for the first level caches 32
Figure 18: Two line sizes a 16M second level cache 33
Figure 19: $CPI_{wtbuffer}$ compared with proportion of writes in Tree. All runs are identical so only one is shown. 34
Figure 20: $CPI_{wtbuffer}$ compared with proportion of writes in Mult. Three runs are shown. 35
Figure 21: $CPI_{wtbuffer}$ vs proportion of writes for all runs of Tree and Mult 36
Figure 22: $CPI_{wtbuffer}$ vs proportion of writes for all runs of TV and SOR 37

Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers

Norman P. Jouppi

March, 1990

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

Projections of computer technology forecast processors with peak performance of 1,000 MIPS in the relatively near future. These processors could easily lose half or more of their performance in the memory hierarchy if the hierarchy design is based on conventional caching techniques. This paper presents hardware techniques to improve the performance of caches.

Miss caching **places a small fully-associative cache between a cache and its refill path. Misses in the cache that hit in the miss cache have only a one cycle miss penalty, as opposed to a many cycle miss penalty without the miss cache. Small miss caches of 2 to 5 entries are shown to be very effective in removing mapping conflict misses in first-level direct-mapped caches.**

Victim caching **is an improvement to miss caching that loads the small fully-associative cache with the victim of a miss and not the requested line. Small victim caches of 1 to 5 entries are even more effective at removing conflict misses than miss caching.**

Stream buffers **prefetch cache lines starting at a cache miss address. The prefetched data is placed in the buffer and not in the cache. Stream buffers are useful in removing capacity and compulsory cache misses, as well as some instruction cache conflict misses. Stream buffers are more effective than previously investigated prefetch techniques at using the next slower level in the memory hierarchy when it is pipelined. An extension to the basic stream buffer, called *multi-way stream buffers*, is introduced. Multi-way stream buffers are useful for prefetching along multiple intertwined data reference streams.**

Together, victim caches and stream buffers reduce the miss rate of the first level in the cache hierarchy by a factor of two to three on a set of six large benchmarks.

Digital Confidential

Copyright © 1990 Digital Equipment Corporation

Table of Contents

1. Introduction 1
2. Baseline Design 2
3. Reducing Conflict Misses: Miss Caching and Victim Caching 5
    3.1. Miss Caching 6
    3.2. Victim Caching 9
    3.3. The Effect of Direct-Mapped Cache Size on Victim Cache Performance 12
    3.4. The Effect of Line Size on Victim Cache Performance 12
    3.5. Victim Caches and Second-Level Caches 13
    3.6. Miss Caches, Victim Caches, and Error Correction 14
4. Reducing Capacity and Compulsory Misses 15
    4.1. Reducing Capacity and Compulsory Misses with Long Lines 15
    4.2. Reducing Capacity and Compulsory Misses with Prefetch Techniques 20
        4.2.1. Stream Buffers 20
        4.2.2. Multi-Way Stream Buffers 24
        4.2.3. Quasi-Sequential Stream Buffers 24
        4.2.4. Stream Buffer Performance vs. Cache Size 27
        4.2.5. Stream Buffer Performance vs. Line Size 27
        4.2.6. Comparison to Classical Prefetch Performance 28
    4.3. Combining Long Lines and Stream Buffers 30
5. Conclusions 32
References 35

List of Figures

Figure 1: Baseline design 2
Figure 2: Baseline design performance 5
Figure 3: Percentage of conflict misses, 4K I and D caches, 16B lines 6
Figure 4: Miss cache organization 7
Figure 5: Conflict misses removed by miss caching 7
Figure 6: Overall cache misses removed by miss caching 8
Figure 7: Victim cache organization 10
Figure 8: Conflict misses removed by victim caching 10
Figure 9: Equivalent set-associativity provided by victim caching 11
Figure 10: Overall cache misses removed by victim caching 12
Figure 11: Victim cache performance with varying direct-mapped data cache size 13
Figure 12: Victim cache performance with varying data cache line size 14
Figure 13: Effect of increasing line size on capacity and compulsory misses 15
Figure 14: Effect of increasing line size on overall miss rate 16
Figure 15: Effect of increasing data cache line size on each benchmark 17
Figure 16: Effect of increasing data cache line size with miss caches 18
Figure 17: Benchmark-specific performance with increasing data cache line size 19
Figure 18: *yacc* and *met* performance with increasing data cache line size 19
Figure 19: Limited time for prefetch 21
Figure 20: Sequential stream buffer design 22
Figure 21: Sequential stream buffer performance 23
Figure 22: Stream buffer bandwidth requirements 23
Figure 23: Four-way stream buffer design 25
Figure 24: Four-way stream buffer performance 26
Figure 25: Quasi-sequential 4-way stream buffer performance 26
Figure 26: Stream buffer performance vs. cache size 27
Figure 27: Stream buffer performance vs. line size 28
Figure 28: System performance with victim cache and stream buffers 33

List of Tables

Table 1: The increasing cost of cache misses 1
Table 2: Test program characteristics 4
Table 3: Baseline system first-level cache miss rates 4
Table 4: Line sizes with minimum miss rates by program 18
Table 5: Upper bound on prefetch performance: percent reduction in misses 29
Table 6: Upper bound of prefetch performance vs. stream buffer performance 29
Table 7: Improvements relative to a 16B instruction line size without miss caching 31
Table 8: Improvements relative to a 16B data line size without miss caching 31
Table 9: Improvements relative to a 16B data line size and 4-entry miss cache 32
Table 10: Effective increase in instruction cache size provided by streambuffer with 16B lines 34
Table 11: Effective increase in data cache size provided with stream buffers and victim caches using 16B lines 34

1. Introduction

Cache performance is becoming increasingly important since it has a dramatic effect on the performance of advanced processors. Table 1 lists some cache miss times and the effect of a miss on machine performance. Over the last decade, cycle time has been decreasing much faster than main memory access time. The average number of machine cycles per instruction has also been decreasing dramatically, especially when the transition from CISC machines to RISC machines is included. These two effects are multiplicative and result in tremendous increases in miss cost. For example, a cache miss on a VAX 11/780 only costs 60% of the average instruction execution. Thus even if every instruction had a cache miss, the machine performance would slow down by only 60%! However, if a RISC machine like the WRL Titan [11] has a miss, the cost is almost ten instruction times. Moreover, these trends seem to be continuing, especially the increasing ratio of memory access time to machine cycle time. In the future a cache miss all the way to main memory on a superscalar machine executing two instructions per cycle could cost well over 100 instruction times! Even with careful application of well-known cache design techniques, machines with main memory latencies of over 100 instruction times can easily lose over half of their potential performance to the memory hierarchy. This makes both hardware and software research on advanced memory hierarchies increasingly important.

Machine	cycles per instr	cycle time (ns)	mem time (ns)	miss cost (cycles)	miss cost (instr)
VAX11/780	10.0	200	1200	6	.6
WRL Titan	1.4	45	540	12	8.6
?	0.5	4	280	70	140.0

Table 1: The increasing cost of cache misses

This paper investigates new hardware techniques for increasing the performance of the memory hierarchy. Section 2 describes a baseline design using conventional caching techniques. The large performance loss due to the memory hierarchy is a detailed motivation for the techniques discussed in the remainder of the paper. Techniques for reducing misses due to mapping conflicts (i.e., lack of associativity) are presented in Section 3. An extension to prefetch techniques called stream buffering is evaluated in Section 4. Section 5 summarizes this work and evaluates promising directions for future work.

2. Baseline Design

Figure 1 shows the range of configurations of interest in this study. The CPU, floating-point unit, memory management unit (e.g., TLB), and first level instruction and data caches are on the same chip or on a single high-speed module built with an advanced packaging technology. (We will refer to the central processor as a single chip in the remainder of the paper, but chip or module is implied.) The cycle time off this chip is 3 to 8 times longer than the instruction issue rate (i.e., 3 to 8 instructions can issue in one off-chip clock cycle). This is obtained either by having a very fast on-chip clock (e.g., superpipelining [9]), by issuing many instructions per cycle (e.g., superscalar or VLIW), and/or by using higher speed technologies for the processor chip than for the rest of the system (e.g., GaAs vs. BiCMOS).

The expected size of the on-chip caches varies with the implementation technology for the processor, but higher-speed technologies generally result in smaller on-chip caches. For example, quite large on-chip caches should be feasible in CMOS but only small caches are feasible in the near term for GaAs or bipolar processors. Thus, although GaAs and bipolar are faster, the higher miss rate from their smaller caches tends to decrease the actual system performance ratio between GaAs or bipolar machines and dense CMOS machines to less than the ratio between their gate speeds. In all cases the first-level caches are assumed to be direct-mapped, since this results in the fastest effective access time [7]. Line sizes in the on-chip caches are most likely in the range of 16B to 32B. The data cache may be either write-through or write-back, but this paper does not examine those tradeoffs.

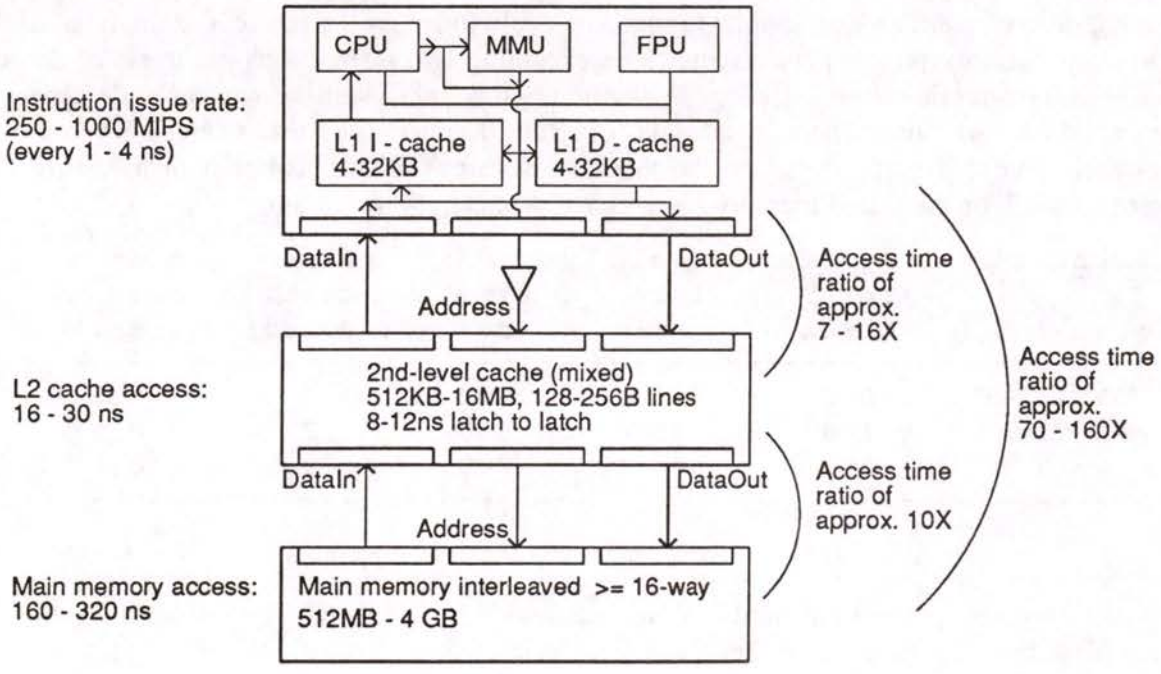


Figure 1: Baseline design

The second-level cache is assumed to range from 512KB to 16MB, and to be built from very high speed static RAMs. It is assumed to be direct-mapped for the same reasons as the first-level

caches. For caches of this size access times of 16 to 30ns are likely. This yields an access time for the cache of 4 to 30 instruction times. The relative speed of the processor as compared to the access time of the cache implies that the second-level cache must be pipelined in order for it to provide sufficient bandwidth. For example, consider the case where the first-level cache is a write-through cache. Since stores typically occur at an average rate of 1 in every 6 or 7 instructions, an unpipelined external cache would not have even enough bandwidth to handle the store traffic for access times greater than seven instruction times. Caches have been pipelined in mainframes for a number of years [8, 13], but this is a recent development for workstations. Recently cache chips with ECL I/O's and registers or latches on their inputs and outputs have appeared; these are ideal for pipelined caches. The number of pipeline stages in a second-level cache access could be 2 or 3 depending on whether the pipestage going from the processor chip to the cache chips and the pipestage returning from the cache chips to the processor are full or half pipestages.

In order to provide sufficient memory for a processor of this speed (e.g., several megabytes per MIP), main memory should be in the range of 512MB to 4GB. This means that even if 16Mb DRAMs are used that it will contain roughly a thousand DRAMs. The main memory system probably will take about ten times longer for an access than the second-level cache. This access time is easily dominated by the time required to fan out address and data signals among a thousand DRAMs spread over many cards. Thus even with the advent of faster DRAMs, the access time for main memory may stay roughly the same. The relatively large access time for main memory in turn requires that second-level cache line sizes of 128 or 256B are needed. As a counter example, consider the case where only 16B are returned after 320ns. This is a bus bandwidth of 50MB/sec. Since a 10 MIP processor with this bus bandwidth would be bus-bandwidth limited in copying from one memory location to another [12], little extra performance would be obtained by the use of a 100 to 1,000 MIP processor. This is an important consideration in the system performance of a processor.

Several observations are in order on the baseline system. First, the memory hierarchy of the system is actually quite similar to that of a machine like the VAX 11/780 [3, 4], only each level in the hierarchy has moved one step closer to the CPU. For example, the 8KB board-level cache in the 780 has moved on-chip. The 512KB to 16MB main memory on early VAX models has become the board-level cache. Just as in the 780's main memory, the incoming transfer size is large (128-256B here vs. 512B pages in the VAX). The main memory in this system is of similar size to the disk subsystems of the early 780's and performs similar functions such as paging and file system caching.

The actual parameters assumed for our baseline system are 1,000 MIPS peak instruction issue rate, separate 4KB first-level instruction and data caches with 16B lines, and a 1MB second-level cache with 128B lines. The miss penalties are assumed to be 24 instruction times for the first level and 320 instruction times for the second level. The characteristics of the test programs used in this study are given in Table 2. These benchmarks are reasonably long in comparison with most traces in use today, however the effects of multiprocessing have not been modeled in this work. The first-level cache miss rates of these programs running on the baseline system configuration are given in Table 3.

The effects of these miss rates are given graphically in Figure 2. The region below the solid line gives the net performance of the system, while the region above the solid line gives the

program name	dynamic instr.	data refs.	total refs.	program type
ccom	31.5M	14.0M	45.5M	C compiler
grr	134.2M	59.2M	193.4M	PC board CAD tool
yacc	51.0M	16.7M	67.7M	Unix utility
met	99.4M	50.3M	149.7M	PC board CAD tool
linpack	144.8M	40.7M	185.5M	numeric, 100x100
liver	23.6M	7.4M	31.0M	LFK (numeric loops)
total	484.5M	188.3M	672.8M	

Table 2: Test program characteristics

program name	baseline miss rate instr.	data
ccom	0.096	0.120
grr	0.061	0.062
yacc	0.028	0.040
met	0.017	0.039
linpack	0.000	0.144
liver	0.000	0.273

Table 3: Baseline system first-level cache miss rates

performance lost in the memory hierarchy. For example, the difference between the top dotted line and the bottom dotted line gives the performance lost due to first-level data cache misses. As can be seen in Figure 2, most benchmarks lose over half of their potential performance in first level cache misses. Only relatively small amounts of performance are lost to second-level cache misses. This is primarily due to the large second-level cache size in comparison to the size of the programs executed. Longer traces [2] of larger programs exhibit significant numbers of second-level cache misses. Since the test suite used in this paper is too small for significant second-level cache activity, second-level cache misses will not be investigated in detail, but will be left to future work.

Since the exact parameters assumed are at the extreme end of the ranges described (maximum performance processor with minimum size caches), other configurations would lose proportionally less performance in their memory hierarchy. Nevertheless, any configuration in the range of interest will lose a substantial proportion of its potential performance in the memory hierarchy. This means that the greatest leverage on system performance will be obtained by improving the memory hierarchy performance, and not by attempting to further increase the performance of the CPU (e.g., by more aggressive parallel issuing of instructions). Techniques for improving the performance of the baseline memory hierarchy at low cost are the subject of the remainder of this paper. Finally, in order to avoid compromising the performance of the CPU core (comprising of the CPU, FPU, MMU, and first level caches), any additional hardware required by the techniques to be investigated should reside outside the CPU core (i.e., below the first level caches). By doing this the additional hardware will only be involved during cache misses, and therefore will not be in the critical path for normal instruction execution.

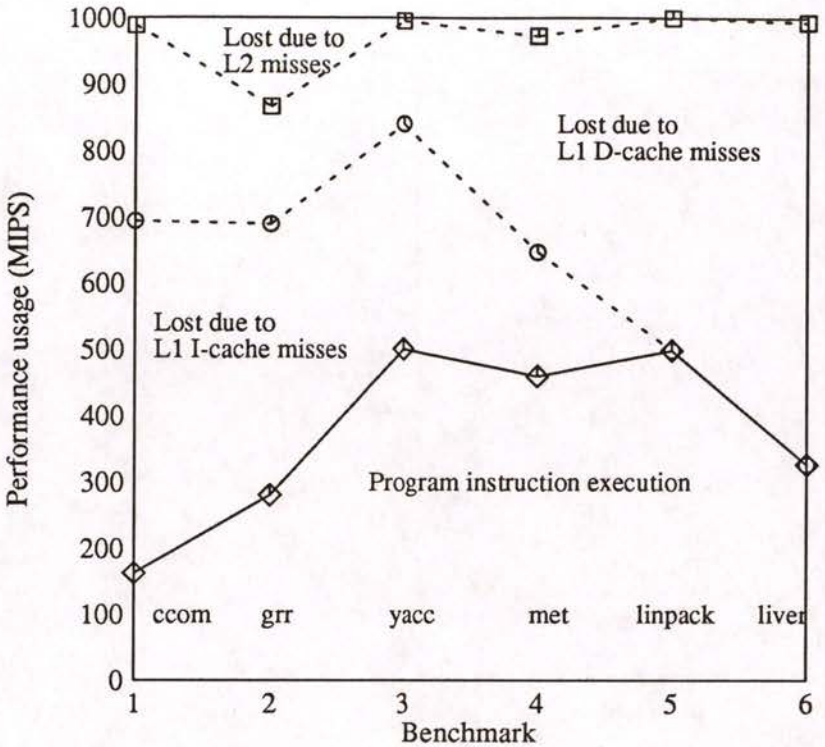


Figure 2: Baseline design performance

3. Reducing Conflict Misses: Miss Caching and Victim Caching

Misses in caches can be classified into four categories: conflict, compulsory, capacity [7], and coherence. Conflict misses are misses that would not occur if the cache was fully-associative and had LRU replacement. Compulsory misses are misses required in any cache organization because they are the first references to an instruction or piece of data. Capacity misses occur when the cache size is not sufficient to hold data between references. Coherence misses are misses that occur as a result of invalidation to preserve multiprocessor cache consistency.

Even though direct-mapped caches have more conflict misses due to their lack of associativity, their performance is still better than set-associative caches when the access time costs for hits are considered. In fact, the direct-mapped cache is the only cache configuration where the critical path is merely the time required to access a RAM [10]. Conflict misses typically account for between 20% and 40% of all direct-mapped cache misses [7]. Figure 3 details the percentage of misses due to conflicts for our test suite. On average 39% of the first-level data cache misses are due to conflicts, and 29% of the first-level instruction cache misses are due to conflicts. Since these are significant percentages, it would be nice to "have our cake and eat it too" by somehow providing additional associativity without adding to the critical access path for a direct-mapped cache.

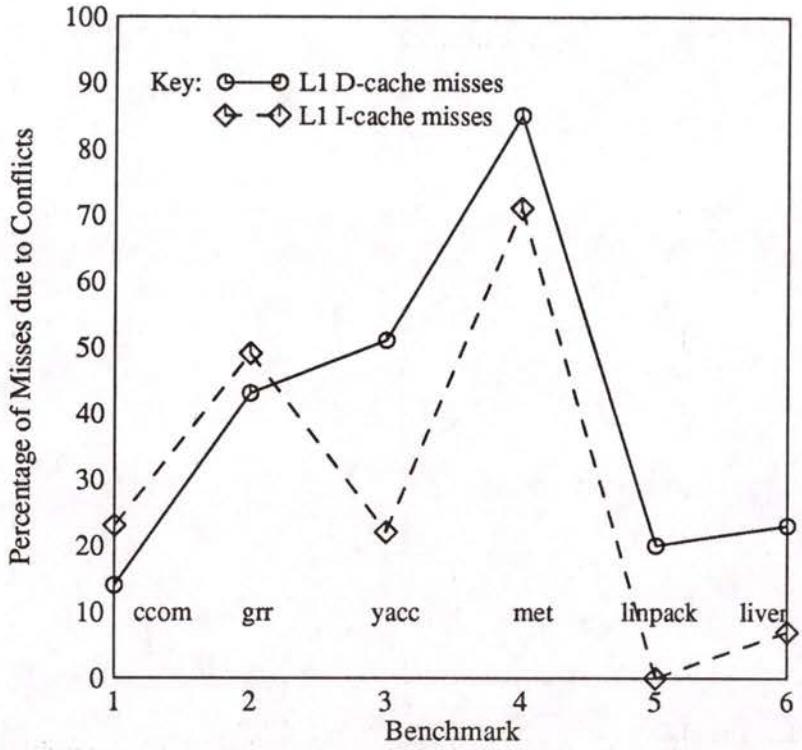


Figure 3: Percentage of conflict misses, 4K I and D caches, 16B lines

3.1. Miss Caching

We can add associativity to a direct-mapped cache by placing a small *miss cache* on-chip between a first-level cache and the access port to the second-level cache (Figure 4). A miss cache is a small fully-associative cache containing on the order of two to five cache lines of data. When a miss occurs, data is returned not only to the direct-mapped cache, but also to the miss cache under it, where it replaces the least recently used item. Each time the upper cache is probed, the miss cache is probed as well. If a miss occurs in the upper cache but the address hits in the miss cache, then the direct-mapped cache can be reloaded in the next cycle from the miss cache. This replaces a long off-chip miss penalty with a short one-cycle on-chip miss. This arrangement satisfies the requirement that the critical path is not worsened, since the miss cache itself is not in the normal critical path of processor execution.

The success of different miss cache organizations at removing conflict misses is shown in Figure 5. The first observation to be made is that many more data conflict misses are removed by the miss cache than instruction conflict misses. This can be explained as follows. Instruction conflicts tend to be widely spaced because the instructions within one procedure will not conflict with each other as long as the procedure size is less than the cache size, which is almost always the case. Instruction conflict misses are most likely when another procedure is called. The target procedure may map anywhere with respect to the calling procedure, possibly resulting in a large overlap. Assuming at least 60 different instructions are executed in each procedure, the conflict misses would span more than the 15 lines in the maximum size miss cache tested. In other words, a small miss cache could not contain the entire overlap and so would be reloaded repeatedly before it could be used. This type of reference pattern exhibits the worst miss cache performance.

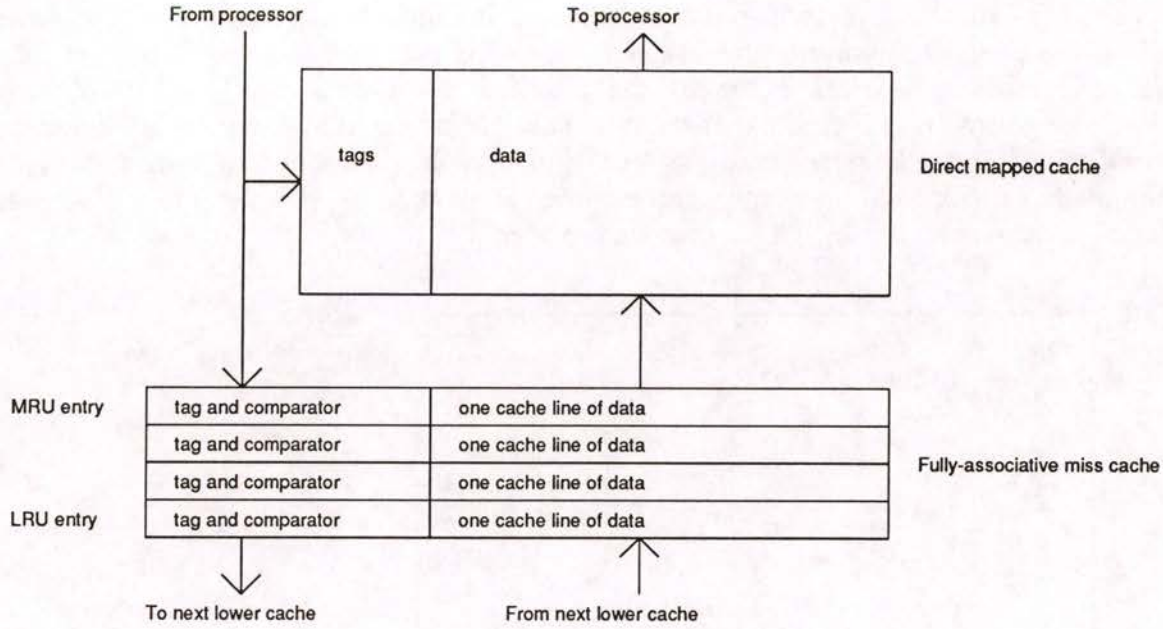


Figure 4: Miss cache organization

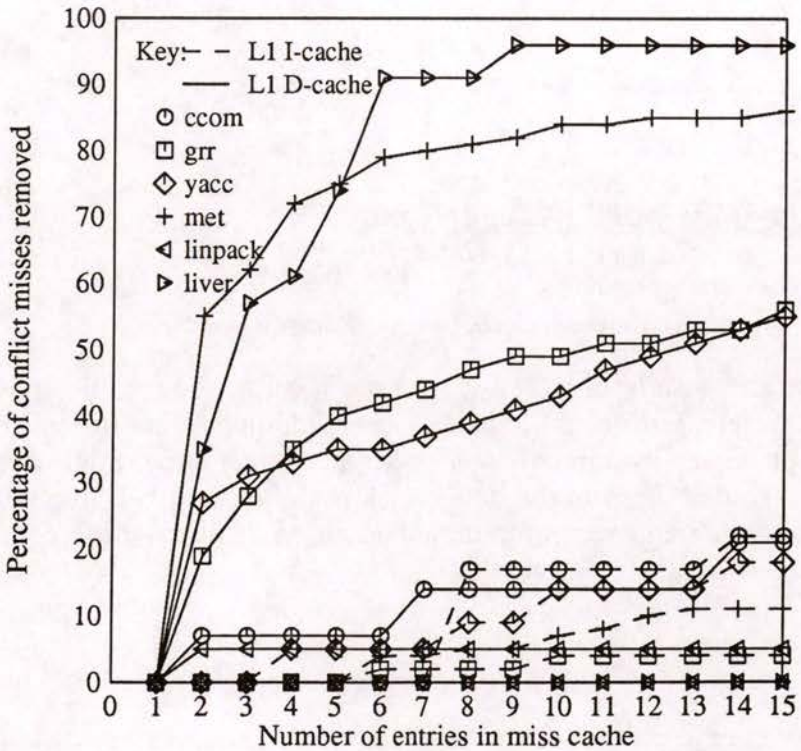


Figure 5: Conflict misses removed by miss caching

Data conflicts, on the other hand, can be quite closely spaced. Consider the case where two character strings are being compared. If the points of comparison of the two strings happen to

map to the same line, alternating references to different strings will always miss in the cache. In this case a miss cache of only two entries would remove all of the conflict misses. Obviously this is another extreme of performance and the results in Figure 5 show a range of performance based on the program involved. Nevertheless, for 4KB data caches a miss cache of only 2 entries can remove 25% percent of the data cache conflict misses on average,* or 13% of the data cache misses overall (see Figure 6). If the miss cache is increased to 4 entries, 36% percent of the conflict misses can be removed, or 18% of the data cache misses overall. After four entries the improvement from additional miss cache entries is minor, only increasing to a 25% overall reduction in data cache misses if 15 entries are provided.

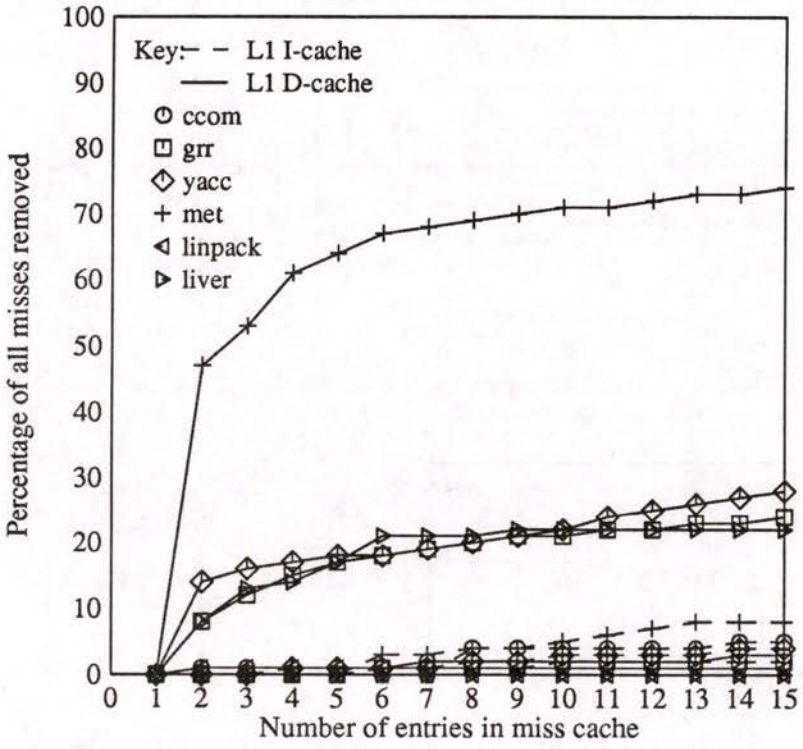


Figure 6: Overall cache misses removed by miss caching

Since doubling the data cache size results in a 32% reduction in misses (over this set of benchmarks when increasing data cache size from 4K to 8K), each additional line in the first level cache reduces the number of misses by approximately 0.13%. Although the miss cache requires more area per bit of storage than lines in the data cache, each line in a two line miss cache effects a 50 times larger marginal improvement in the miss rate, so this should more than cover any differences in layout size.

*Throughout this paper the average reduction in miss rates is used as a metric. This is computed by calculating the percent reduction in miss rate for each benchmark, and then taking the average of these percentages. This has the advantage that it is independent of the number of memory references made by each program. Furthermore, if two programs have widely different miss rates, the average percent reduction in miss rate gives equal weighting to each benchmark. This is in contrast with the percent reduction in average miss rate, which weights the program with the highest miss rate most heavily.

Comparing Figure 5 and Figure 3, we see that the higher the percentage of misses due to conflicts, the more effective the miss cache is at eliminating them. For example, in Figure 3 *met* has by far the highest ratio of conflict misses to total data cache misses. Similarly, *grr* and *yacc* also have greater than average percentages of conflict misses, and the miss cache helps these programs significantly as well. *linpack* and *ccom* have the lowest percentage of conflict misses, and the miss cache removes the lowest percentage of conflict misses from these programs. This results from the fact that if a program has a large percentage of data conflict misses then they must be clustered to some extent because of their overall density. This does not prevent programs with a small number of conflict misses such as *liver* from benefiting from a miss cache, but it seems that as the percentage of conflict misses increases, the percentage of these misses removable by a miss cache increases.

3.2. Victim Caching

Consider a system with a direct-mapped cache and a miss cache. When a miss occurs, data is loaded into both the miss cache and the direct-mapped cache. In a sense, this duplication of data wastes storage space in the miss cache. The number of duplicate items in the miss cache can range from one (in the case where all items in the miss cache map to the same line in the direct-mapped cache) to all of the entries (in the case where a series of misses occur which do not hit in the miss cache).

To make better use of the miss cache we can use a different replacement algorithm for the small fully-associative cache [5]. Instead of loading the requested data into the miss cache on a miss, we can load the fully-associative cache with the victim line from the direct-mapped cache instead. We call this *victim caching* (see Figure 7). With victim caching, no data line appears both in the direct-mapped cache and the victim cache. This follows from the fact that the victim cache is loaded only with items thrown out from the direct-mapped cache. In the case of a miss in the direct-mapped cache that hits in the victim cache, the contents of the direct-mapped cache line and the matching victim cache line are swapped.

Depending on the reference stream, victim caching can either be a small or significant improvement over miss caching. The magnitude of this benefit depends on the amount of duplication in the miss cache. Victim caching is always an improvement over miss caching.

As an example, consider an instruction reference stream that calls a small procedure in its inner loop that conflicts with the loop body. If the total number of conflicting lines between the procedure and loop body were larger than the miss cache, the miss cache would be of no value since misses at the beginning of the loop would be flushed out by later misses before execution returned to the beginning of the loop. If a victim cache is used instead, however, the number of conflicts in the loop that can be captured is doubled compared to that stored by a miss cache. This is because one set of conflicting instructions lives in the direct-mapped cache, while the other lives in the victim cache. As execution proceeds around the loop and through the procedure call these items trade places.

The percentage of conflict misses removed by victim caching is given in Figure 8. Note that victim caches consisting of just one line are useful, in contrast to miss caches which must have two lines to be useful. All of the benchmarks have improved performance in comparison to miss caches, but instruction cache performance and the data cache performance of benchmarks that have conflicting long sequential reference streams (e.g., *ccom* and *linpack*) improve the most.

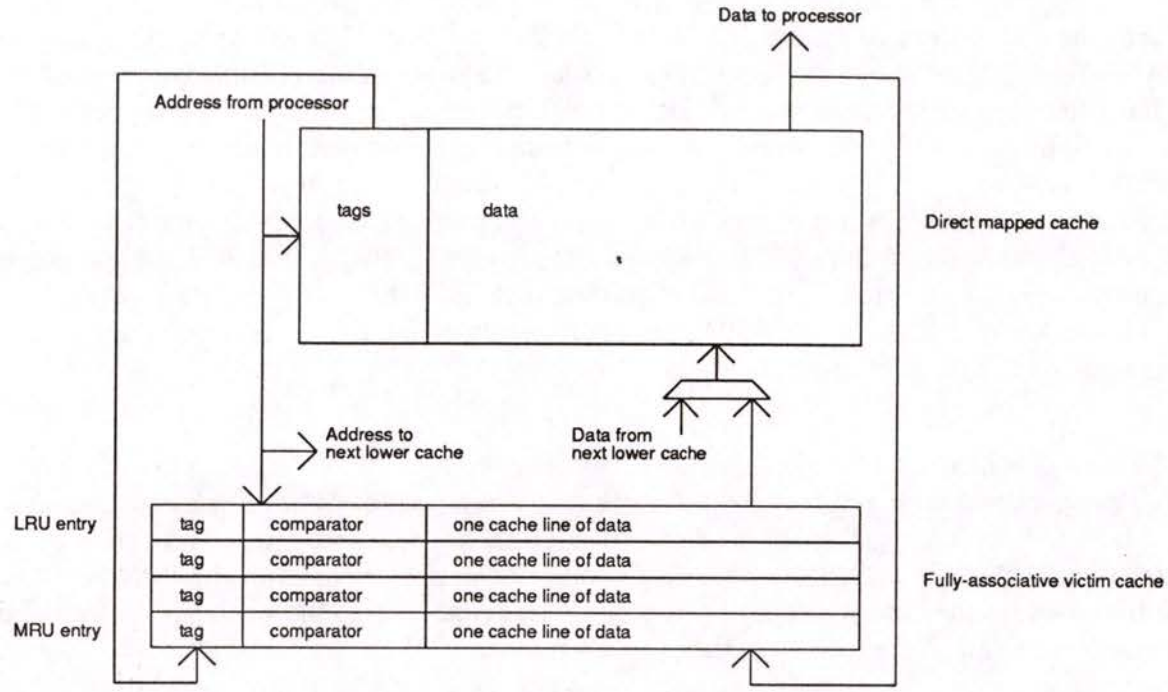


Figure 7: Victim cache organization

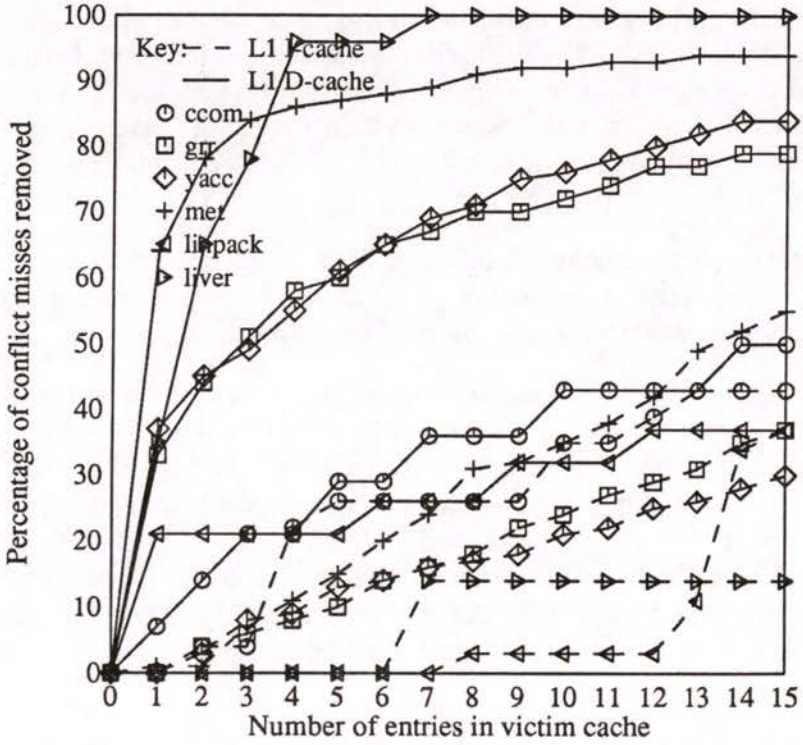


Figure 8: Conflict misses removed by victim caching

The reduction in conflict misses obtained by victim caching is shown in Figure 9 relative to the performance of a 2-way set associative cache (the key is the same as for Figure 8). Note that

a one-entry victim cache provides about 50% of the benefit of 2-way set-associativity for *liver*, *met*, and *grr*. In fact, a direct-mapped cache with a 2-entry victim cache performs better than a 2-way set associative cache on *liver*. The dashed line for *linpack* instruction references at a victim cache size of seven results from the fact that a 2-way set-associative instruction cache performs slightly worse for *linpack* than a direct-mapped cache, and a direct-mapped cache with an 8-entry victim cache performs slightly better than a direct-mapped cache alone.

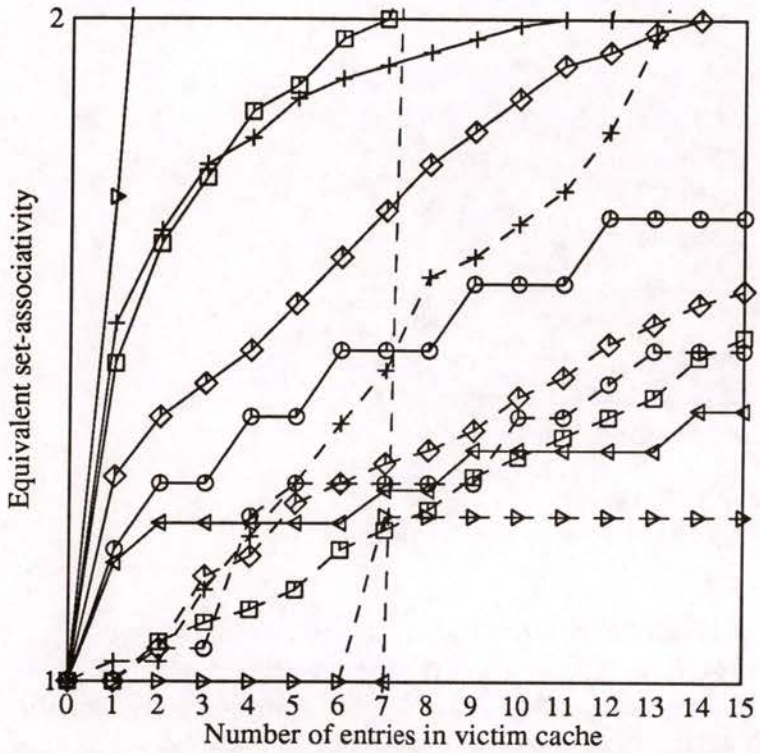


Figure 9: Equivalent set-associativity provided by victim caching

Figure 10 shows the overall reduction in miss rate possible with victim caching. As can be seen by comparing Figure 6 and Figure 10, the performance of the victim cache is in some cases better than a miss cache with twice the number of entries. For example, consider *yacc*'s data cache performance with a one-entry victim cache and a two-entry miss cache. Because the victim cache doesn't throw away the victim, in some situations victim caching can result in fewer misses than a miss cache with twice the number of entries. For example, imagine many cache misses occur accessing new data (i.e., compulsory misses), effectively flushing out both a miss cache and a victim cache. Next imagine another new line is referenced, causing a miss for both a system with a miss cache and a system with a victim cache. If the old contents of the line are referenced next, the miss cache will not contain the item, but a victim cache would. Thus the system with a miss cache would have two misses to the next level in the memory hierarchy, while the system with a victim cache would only have one.

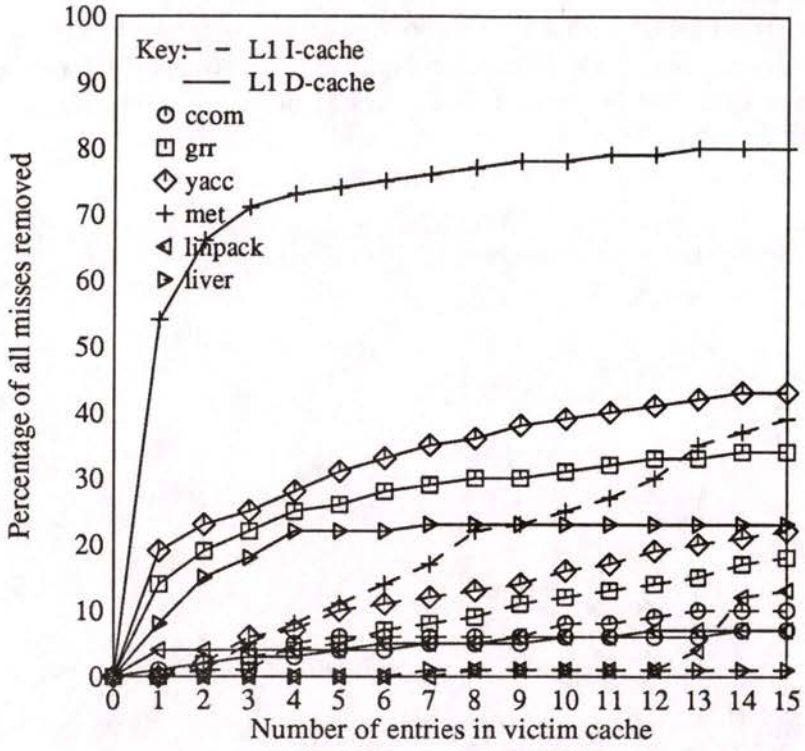


Figure 10: Overall cache misses removed by victim caching

3.3. The Effect of Direct-Mapped Cache Size on Victim Cache Performance

Figure 11 shows the performance of 1, 2, 4, and 15 entry victim caches when backing up direct-mapped data caches of varying sizes. In general smaller direct-mapped caches benefit the most from the addition of a victim cache. Also shown for reference is the total percentage of conflict misses for each cache size. There are two factors to victim cache performance versus direct-mapped cache size. First, as the direct-mapped cache increases in size, the relative size of the victim cache becomes smaller. Since the direct-mapped cache gets larger but keeps the same line size (16B), the likelihood of a tight mapping conflict which would be easily removed by victim caching is reduced. Second, the percentage of conflict misses decreases slightly from 1KB to 32KB. As we have seen previously, as the percentage of conflict misses decreases, the percentage of these misses removed by the victim cache decreases. The first effect dominates, however, since as the percentage of conflict misses increases with very large caches (as in [7]), the victim cache performance only improves slightly.

3.4. The Effect of Line Size on Victim Cache Performance

Figure 12 shows the performance of victim caches for 4KB direct-mapped data caches of varying line sizes. As one would expect, as the line size at this level increases, the number of conflict misses also increases. The increasing percentage of conflict misses results in an increasing percentage of these misses being removed by the victim cache. Systems with victim caches can benefit from longer line sizes more than systems without victim caches, since the victim caches help remove misses caused by conflicts that result from longer cache lines. Note that even if the area used for data storage in the victim cache is held constant (i.e., the number of

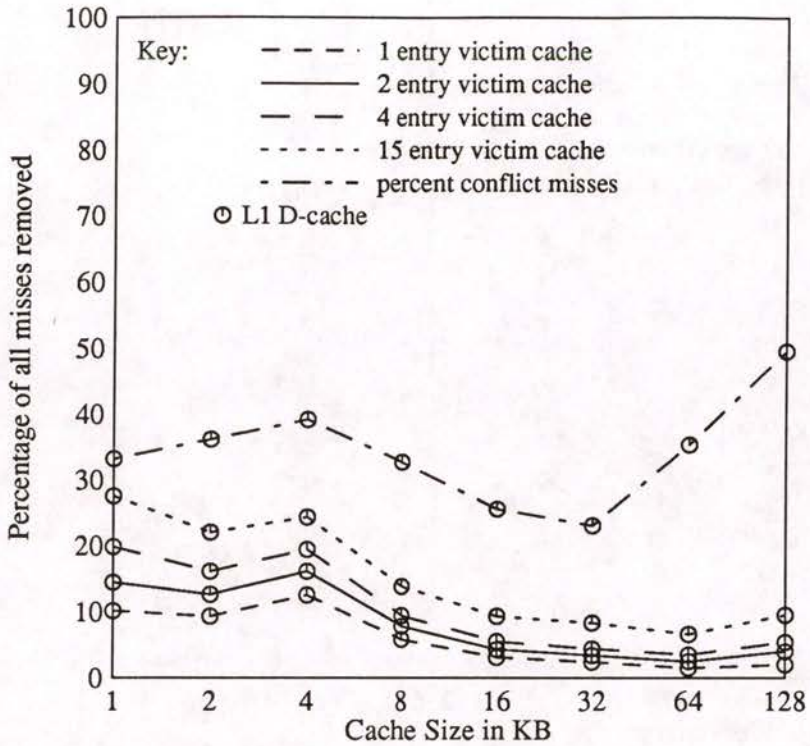


Figure 11: Victim cache performance with varying direct-mapped data cache size

entries is cut in half when the line size doubles) the performance of the victim cache still improves or at least breaks even when line sizes increase.

3.5. Victim Caches and Second-Level Caches

As the size of a cache increases, a larger percentage of its misses are due to conflict and compulsory misses and fewer are due to capacity misses. (Unless of course the cache is larger than the entire program, in which case only compulsory misses remain.) Thus victim caches might be expected to be useful for second-level caches as well. Since the number of conflict misses increases with increasing line sizes, the large line sizes of second-level caches would also tend to increase the potential usefulness of victim caches.

One interesting aspect of victim caches is that they violate inclusion properties [1] in cache hierarchies. However, the line size of the second level cache in the baseline design is 8 to 16 times larger than the first-level cache line sizes, so this violates inclusion as well.

Note that a first-level victim cache can contain many lines that conflict not only at the first level but also at the second level. Thus, using a first-level victim cache can also reduce the number of conflict misses at the second level. In investigating victim caches for second-level caches, both configurations with and without first-level victim caches will need to be considered.

A thorough investigation of victim caches for megabyte second-level caches requires traces of billions of instructions. At this time we only have victim cache performance for our smaller test suite, and work on obtaining victim cache performance for multi-megabyte second-level caches is underway.

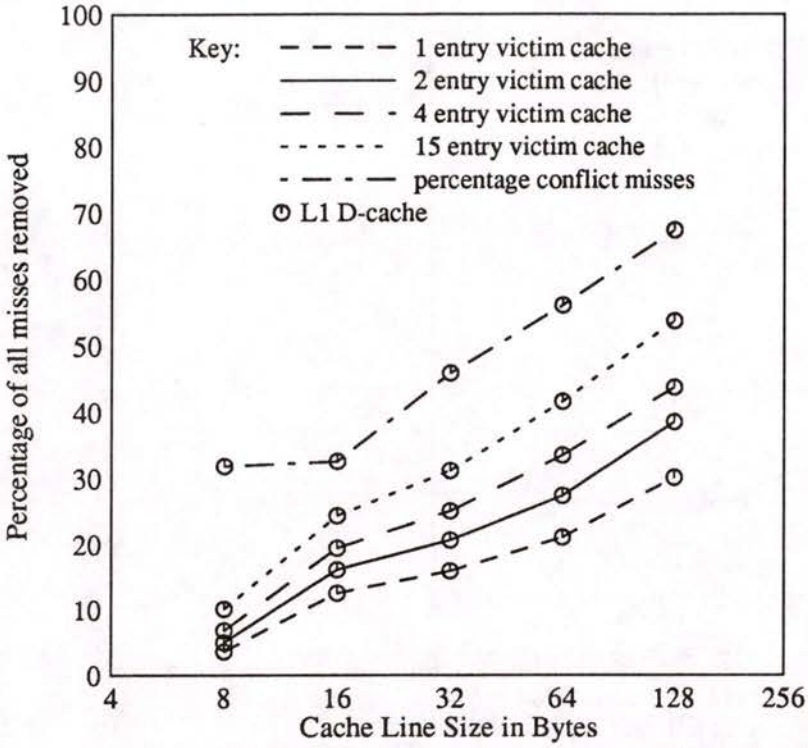


Figure 12: Victim cache performance with varying data cache line size

3.6. Miss Caches, Victim Caches, and Error Correction

Another important use for miss caches, especially on-chip at the first-level, is in yield enhancement and fault tolerance. If parity is kept on all instruction and data cache bytes, and the data cache is write-though, then cache parity errors can be handled as misses. If the refill path bypasses the cache, then this scheme can also allow chips with hard errors to be used. (In fact with byte parity, up to 1/9 of all bits in the cache could be faulty as long as there were at most one bad bit per byte.) Unfortunately, without miss caches if the inner loop of *linpack* (i.e., saxpy) happens to land on a line with a defect or if a frequently used structure variable is on a defective line, the performance of the system can be severely degraded (e.g., by greater than a factor of four on some code segments). Moreover the performance degradation would vary from chip to chip seemingly at random depending on defect location. This would limit the potential yield enhancement to the engineering development phase of a project. However, with the addition of miss caches, the penalty on a defect-induced parity miss is only one cycle, which would have a much smaller impact on machine performance than an off-chip miss. Thus, as long as the number of defects was small enough to be handled by the miss cache, chips with hard defects could be used in production systems. If miss caches are used to improve system performance in the presence of fabrication defects, then instruction miss caches and even miss caches with only one entry would be useful.

Victim caches as described earlier would not be useful for correction of misses due to parity errors. This is because the victim is corrupted by the parity error, and is not worth saving. However victim caches can also be used for error-correction with the following change. When a cache miss is caused by a parity error, the victim cache is loaded with the incoming (miss) data

and not the victim. Thus it acts like a victim cache for normal misses and a miss cache for parity misses. With this minor modification the benefits of miss caches for error-recovery and the better performance of victim caching can be combined.

4. Reducing Capacity and Compulsory Misses

Compulsory misses are misses required in any cache organization because they are the first references to a piece of data. Capacity misses occur when the cache size is not sufficient to hold data between references. One way of reducing the number of capacity and compulsory misses is to use prefetch techniques such as longer cache line sizes or prefetching methods [14, 6]. However, line sizes can not be made arbitrarily large without increasing the miss rate and greatly increasing the amount of data to be transferred. In this section we investigate techniques to reduce capacity and compulsory misses while mitigating traditional problems with long lines and excessive prefetching.

4.1. Reducing Capacity and Compulsory Misses with Long Lines

If conflict misses did not exist, caches with much larger line sizes would be appropriate. Figure 13 shows the reduction in compulsory and capacity misses with increasing line size, compared to a baseline design with 8B lines. In general, all benchmarks have reduced miss rates as the line size is increased, although *yacc* has anomalous instruction cache behavior at 64B line sizes.

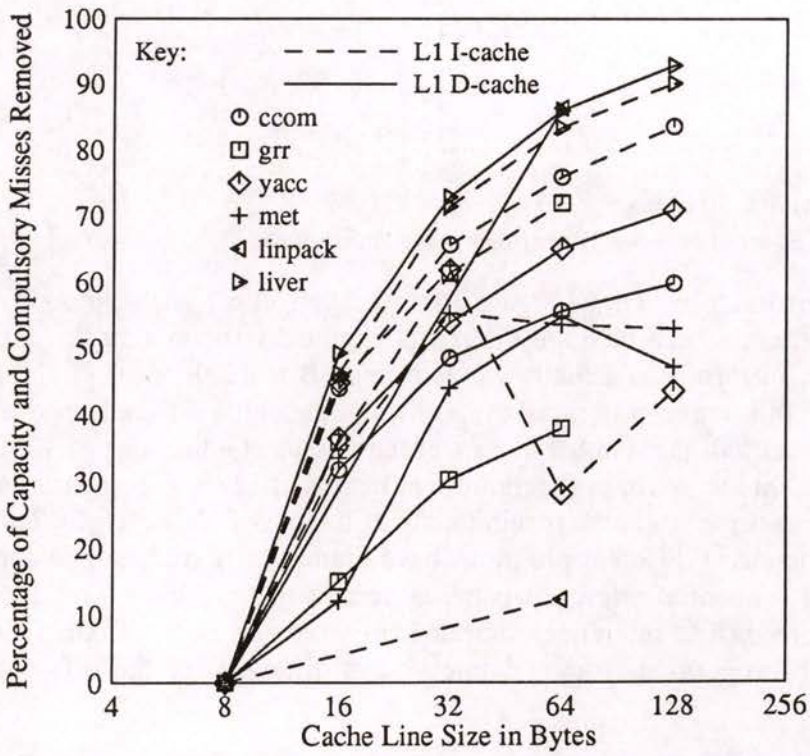


Figure 13: Effect of increasing line size on capacity and compulsory misses

However, when the effects of conflict misses are included, the picture changes dramatically (see Figure 14). As can be seen, the instruction cache performance still increases with increasing

line size but the data cache performance peaks at a modest line size and decreases for further increases in line size beyond that. This is a well known effect and is due to differences in spatial locality between instruction and data references. For example, when a procedure is called, many instructions within a given extent will be executed. However, data references tend to be much more scattered, especially in programs that are not based on unit-stride array access. Thus the long line sizes are much more beneficial to quasi-sequential instruction access patterns than the more highly distributed data references.

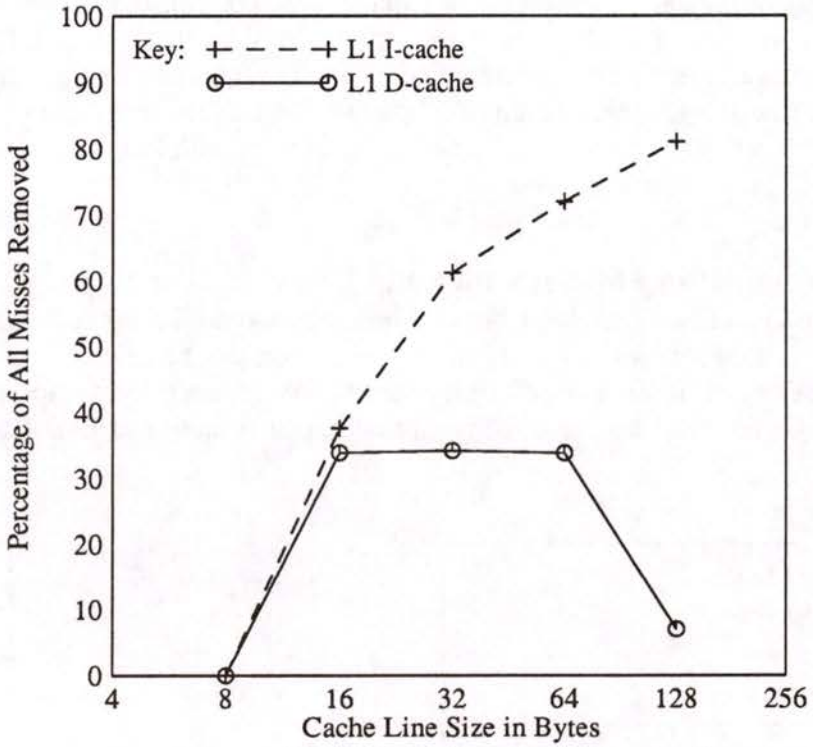


Figure 14: Effect of increasing line size on overall miss rate

Although curves of average performance such as Figure 14 appear to roll off fairly smoothly, the performance for individual programs can be quite different. Figure 15 shows that the data cache line size providing the best performance actually varies from 16B to 128B, depending on the program. Moreover, within this range programs can have dramatically different performance. For example, *liver* has about half the number of data cache misses at a line size of 128B as compared to 16B, but *met* has about three times the number of misses at 128B as compared to 16B. Similarly the performance of *yacc* degrades precipitously at line sizes above 16B. This shows one problem with large line sizes: different programs have dramatically different performance. For programs with long sequential reference patterns, relatively long lines would be useful, but for programs with more diffuse references shorter lines would be best. Taking it a step further, even within a given program the optimal line size is different for the different references that a program makes.

Since the performance in Figure 13 increases fairly monotonically with increasing line size, we know the steep drops in performance in Figure 15 are due to increasing numbers of conflict misses. Since miss caches tend to remove a higher percentage of conflict misses when conflicts

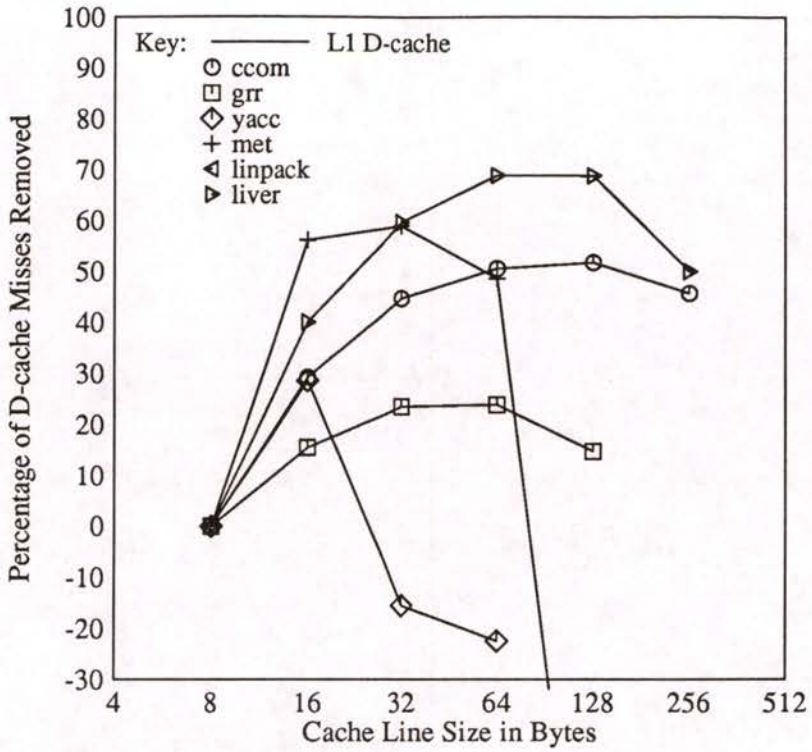


Figure 15: Effect of increasing data cache line size on each benchmark

are frequent, miss caches should allow us to take better advantage of longer cache line sizes. Figure 16 shows the average effectiveness of increasing line size in configurations with and without miss caches. By adding a miss cache more benefits can be derived from a given increase in line size, as well increasing the line size at which the minimum miss rate occurs. This effect can be quite significant: increasing the line size from 16B to 32B with a 4-entry miss cache decreases the miss rate by 36.3%, but only decreases it by 0.5% on average when increasing the line size without a miss cache. Table 4 shows the minimum miss rate for each benchmark with and without miss caches. Benchmarks with minimum miss rate line sizes that are not powers of two have equal miss rates at the next larger and smaller powers of two. The geometric mean over the six benchmarks of the line size giving the lowest miss rate increases from 46B to 92B with the addition of a 4-entry miss cache. The minimum line size giving the best performance on any of the six benchmarks also increases from 16B to 32B with the addition of a 4-entry miss cache.

Figure 17 shows the detailed behavior of most of the programs. The performance of systems with 8B lines are all normalized to zero, independent of the size of their miss cache (if any). This removes the reduction in misses simply due to miss caching from the comparison of the effects of longer cache lines. Thus the actual performance of systems with miss caches at 16B lines are all better than systems without miss caches.

Systems with miss caching continue to obtain benefits from longer line sizes where systems without miss caches have flat or decreasing performance. Figure 18 shows the effects of longer cache line sizes on *yacc* and *met* with varying miss cache sizes, similarly normalized to performance with 8B lines. The performance of *yacc* is affected most dramatically - the sharp dropoff

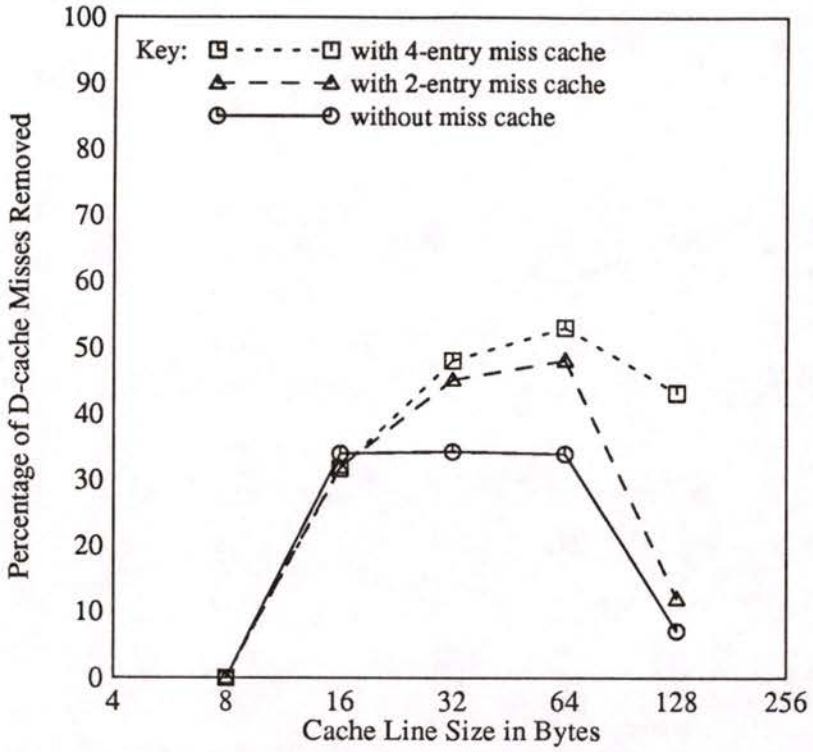


Figure 16: Effect of increasing data cache line size with miss caches

miss cache entries	line size with minimum miss rate					geom mean	min
	ccom	grr	yacc	met	liver		
4	256	96	64	32	128	92	32
2	128	64	128	32	128	84	32
0	128	48	16	32	64	46	16

Table 4: Line sizes with minimum miss rates by program

at line sizes above 16B is completely eliminated even with miss caches with as few as two entries. The performance of *met* is a little more subtle. A system with a miss cache, although always performing better than a system without a miss cache, does not benefit as much on *met* from an increase in line size. Thus the number of additional misses removed with longer lines when using miss caches for *met* is lower than when not using a miss cache for line sizes in the range of 16B to 64B. However the absolute miss rate (not shown) is still lower when using the miss caches. At line sizes of 128B, adding a miss cache with four entries can turn a 100% increase in miss rate for *met* into only a 22% increase in miss rate, although a two entry miss cache has little effect. This benchmark is the primary reason why the average performance of two-entry and four-entry miss caches in Figure 16 diverge at a line size of 128B.

Miss caches for very large lines or with more than four entries at moderate line sizes were not simulated. As line sizes become larger, the amount of storage required by the miss cache increases dramatically: with our 4KB cache an 8-entry miss cache with 128B lines requires an amount of storage equal to 1/4 the total cache size! An interesting area of future research for

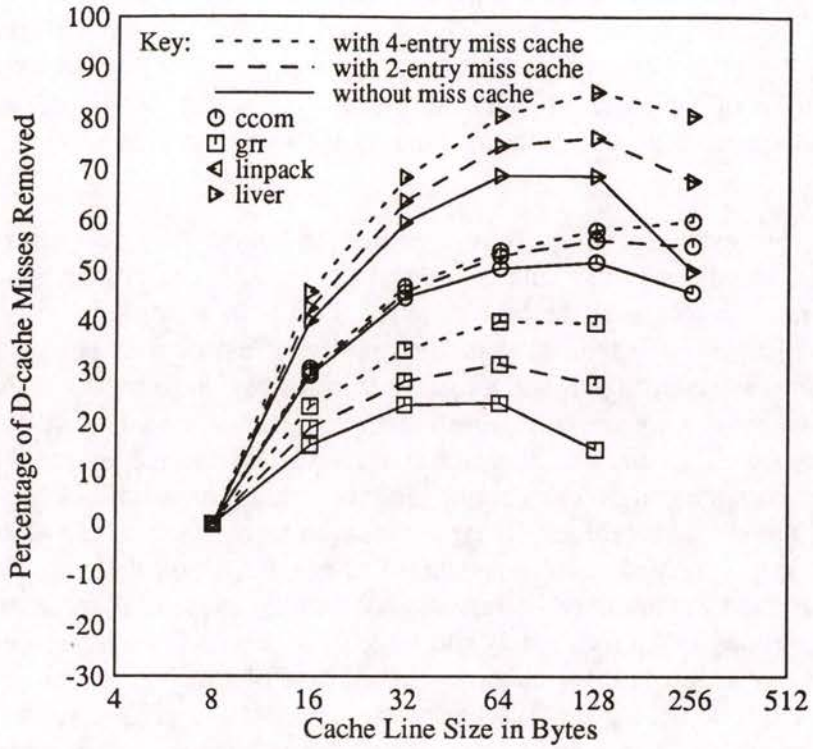


Figure 17: Benchmark-specific performance with increasing data cache line size

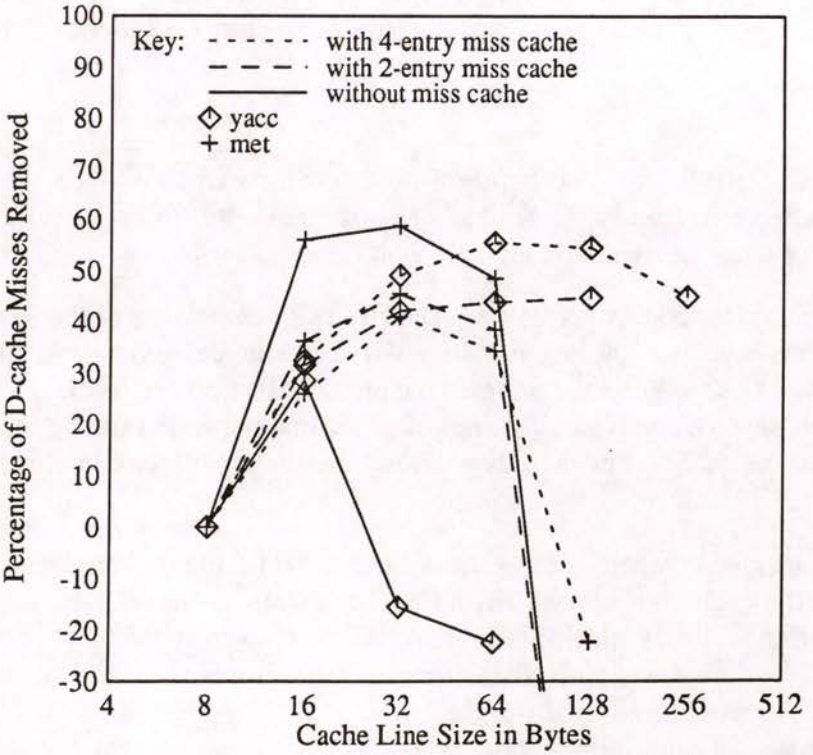


Figure 18: *yacc* and *met* performance with increasing data cache line size

systems with very long lines is the possibility of miss caching on subblocks. Much of the benefit of full-line miss caches might then be obtained with a fraction of the storage requirements.

4.2. Reducing Capacity and Compulsory Misses with Prefetch Techniques

Longer line sizes suffer from the disadvantage of providing a fixed transfer size for different programs and access patterns. Prefetch techniques are interesting because they can be more adaptive to the actual access patterns of the program. This is especially important for improving the performance on long quasi-sequential access patterns such as instruction streams or unit-stride array accesses.

A detailed analysis of three prefetch algorithms has appeared in [14]. *Prefetch always* prefetches after every reference. Needless to say this is impractical in our base system since many level-one cache accesses can take place in the time required to initiate a single level-two cache reference. This is especially true in machines that fetch multiple instructions per cycle from an instruction cache and can concurrently perform a load or store per cycle to a data cache. *Prefetch on miss* and *tagged prefetch* are more promising techniques. On a miss *prefetch on miss* always fetches the next line as well. It can cut the number of misses for a purely sequential reference stream in half. *Tagged prefetch* can do even better. In this technique each block has a tag bit associated with it. When a block is prefetched, its tag bit is set to zero. Each time a block is used its tag bit is set to one. When a block undergoes a zero to one transition its successor block is prefetched. This can reduce the number of misses in a purely sequential reference stream to zero, if fetching is fast enough. Unfortunately the large latencies in the base system can make this impossible. Consider Figure 19, which gives the amount of time (in instruction issues) until a prefetched line is required during the execution of *ccom*. Not surprisingly, since the line size is four instructions, prefetched lines must be received within four instruction-times to keep up with the machine on uncached straight-line code. Because the base system second-level cache takes many cycles to access, and the machine may actually issue many instructions per cycle, tagged prefetch may only have a one-cycle-out-of-many head start on providing the required instructions.

4.2.1. Stream Buffers

What we really need to do is to start the prefetch before a tag transition can take place. We can do this with a mechanism called a *stream buffer* (Figure 20). A stream buffer consists of a series of entries, each consisting of a tag, an available bit, and a data line.

When a miss occurs, the stream buffer begins prefetching successive lines starting at the miss target. As each prefetch request is sent out, the tag for the address is entered into the stream buffer, and the available bit is set to false. When the prefetch data returns it is placed in the entry with its tag and the available bit is set to true. Note that lines after the line requested on the miss are placed in the buffer and not in the cache. This avoids polluting the cache with data that may never be needed.

Subsequent accesses to the cache also compare their address against the first item stored in the buffer. If a reference misses in the cache but hits in the buffer the cache can be reloaded in a single cycle from the stream buffer. This is much faster than the off-chip miss penalty. The stream buffers considered in this section are simple FIFO queues, where only the head of the queue has a tag comparator and elements removed from the buffer must be removed strictly in sequence without skipping any lines. In this simple model non-sequential line misses will cause a stream buffer to be flushed and restarted at the miss address even if the requested line is already present further down in the queue. More complicated stream buffers that can provide already-fetched lines out of sequence are discussed in following sections.

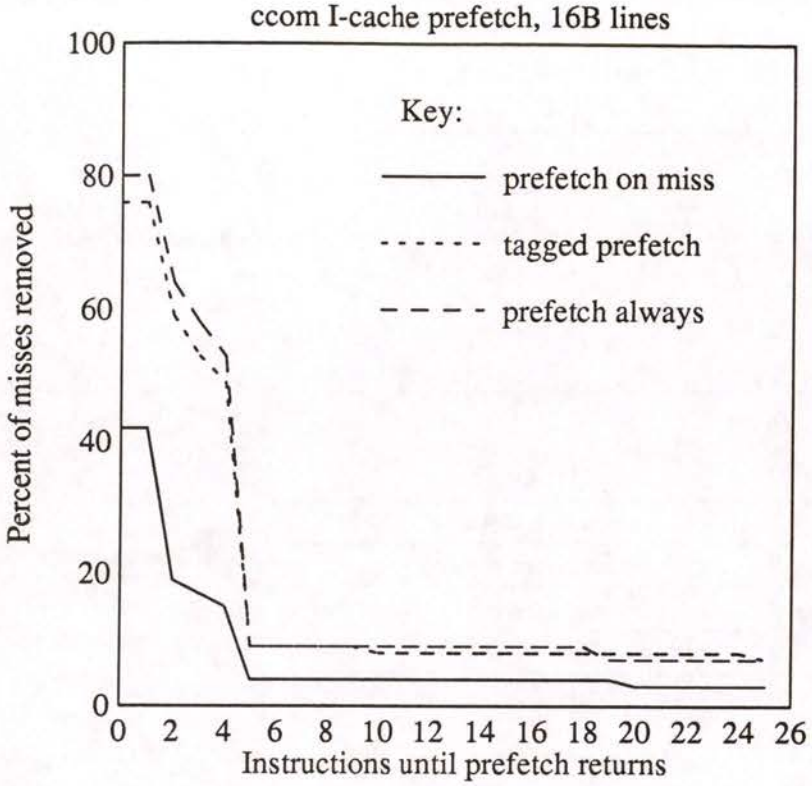


Figure 19: Limited time for prefetch

When a line is moved from a stream buffer to the cache, the entries in the stream buffer can shift up by one and a new successive address is fetched. The pipelined interface to the second level allows the buffer to be filled at the maximum bandwidth of the second level cache, and many cache lines can be in the process of being fetched simultaneously. For example, assume the latency to refill a 16B line on a instruction cache miss is 12 cycles. Consider a memory interface that is pipelined and can accept a new line request every 4 cycles. A four-entry stream buffer can provide 4B instructions at a rate of one per cycle by having three requests outstanding at all times. Thus during sequential instruction execution long latency cache misses will not occur. This is in contrast to the performance of tagged prefetch on purely sequential reference streams where only one line is being prefetched at a time. In that case sequential instructions will only be supplied at a bandwidth equal to one instruction every three cycles (i.e., 12 cycle latency / 4 instructions per line).

Figure 21 shows the performance of a four-entry instruction stream buffer backing a 4KB instruction cache and a data stream buffer backing a 4KB data cache, each with 16B lines. The graph gives the cumulative number of misses removed based on the number of lines that the buffer is allowed to prefetch after the original miss. (In practice the stream buffer would probably be allowed to fetch until the end of a virtual memory page or a second-level cache line. The major reason for plotting stream buffer performance as a function of prefetch length is to get a better idea of how far streams continue on average.) Most instruction references break the purely sequential access pattern by the time the 6th successive line is fetched, while many data reference patterns end even sooner. The exceptions to this appear to be instruction references for *liver* and data references for *linpack*. *liver* is probably an anomaly since the 14 loops of the program are executed sequentially, and the first 14 loops do not generally call other procedures

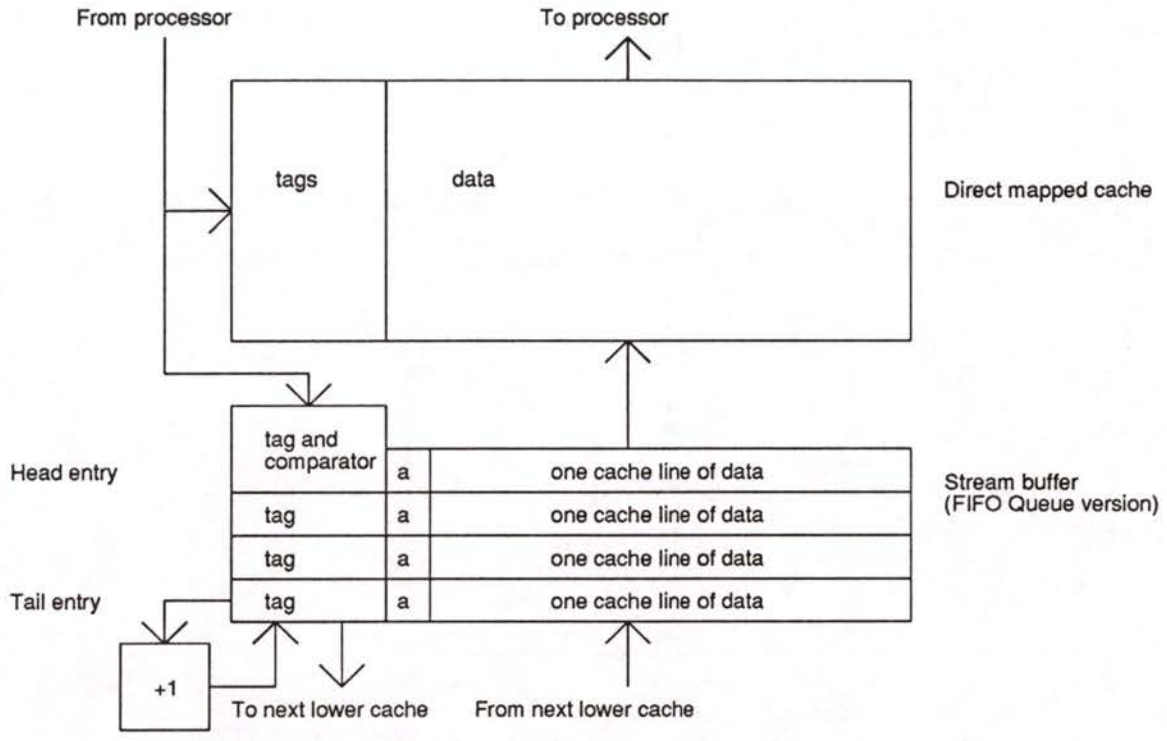


Figure 20: Sequential stream buffer design

or do excessive branching, which would cause the sequential miss pattern to break. The data reference pattern of *linpack* can be understood as follows. Remember that the stream buffer is only responsible for providing lines that the cache misses on. The inner loop of *linpack* (i.e., saxpy) performs an inner product between one row and the other rows of a matrix. The first use of the one row loads it into the cache. After that subsequent misses in the cache (except for mapping conflicts with the first row) consist of subsequent lines of the matrix. Since the matrix is too large to fit in the on-chip cache, the whole matrix is passed through the cache on each iteration. The stream buffer can do this at the maximum bandwidth provided by the second-level cache. Of course one prerequisite for this is that the reference stream is unit-stride or at most skips to every other or every third word. If an array is accessed in the non-unit-stride direction (and the other dimensions have non-trivial extents) then a stream buffer as presented here will be of little benefit.

Figure 22 gives the bandwidth requirements in three typical stream buffer applications. I-stream references for *ccom* are quite regular (when measured in instructions). On average a new 16B line must be fetched every 4.2 instructions. The spacing between references to the stream buffer increases when the program enters short loops and decreases when the program takes small forward jumps, such as when skipping an else clause. Nevertheless the fetch frequency is quite regular. This data is for a machine with short functional unit latencies, such as the MIPS R2000 or the MultiTitan CPU, so the CPI is quite close to 1 without cache misses.

Data stream buffer reference timings for *linpack* and *ccom* are also given in Figure 22. The reference rate for new 16B lines for *linpack* averages one every 27 instructions. Since this ver-

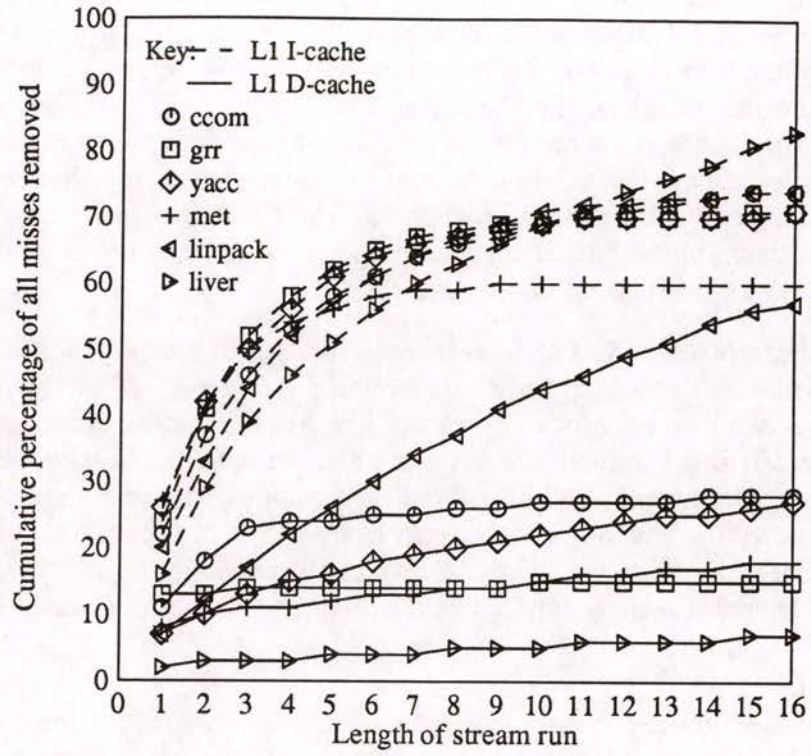


Figure 21: Sequential stream buffer performance

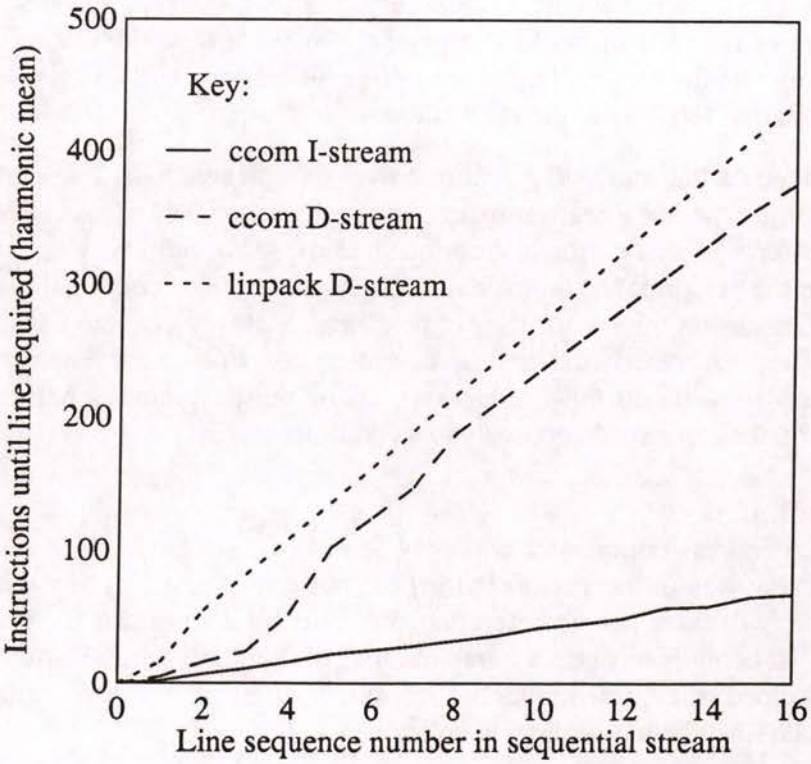


Figure 22: Stream buffer bandwidth requirements

sion of *linpack* is double-precision, this works out to a new iteration of the inner loop every 13.5

instructions. This is larger than one would hope. This version of *linpack* is rather loose in that it does an integer multiply for addressing calculations for each array element, and the loop is not unrolled. If the loop were unrolled and extensive optimizations were performed the rate of references would increase, but the rate should still be less than that of the instruction stream. *ccom* has interesting trimodal performance. If the next successive line is used next after a miss it is required on average only 5 cycles after the miss. For the next two lines after a miss, successive data lines (16B) are required every 10 instructions on average. The first three lines provide most (82%) of the benefit of the stream buffer. After that successive lines are required at a rate closer to that of *linpack*, about every 24 instructions on average.

In general, if the backing store can produce data at an average bandwidth of a new word (4B) every cycle, the stream buffer will be able to keep up with successive references. This should suffice for instruction streams, as well as for block copies that are heavily unrolled and use double-precision loads and stores. If this bandwidth is not available, the benefit of instruction stream buffers will be reduced and block copies and other similar operations will be negatively impacted as well. However, bandwidths equaling a new word every 1.5 to 2 cycles will still suffice for many of the data references. Note that these values are for bandwidths, which are much easier to achieve than total latencies such as required by the prefetch schemes in Figure 19.

4.2.2. Multi-Way Stream Buffers

Overall, the stream buffer presented in the previous section could remove 72% of the instruction cache misses, but it could only remove 25% of the data cache misses. One reason for this is that data references tend to consist of interleaved streams of data from different sources. In order to improve the performance of stream buffers for data references, a multi-way stream buffer was simulated (Figure 23). It consists of four stream buffers in parallel. When a miss occurs in the data cache that does not hit in any stream buffer, the stream buffer hit least recently is cleared (i.e., LRU replacement) and it is started fetching at the miss address.

Figure 24 shows the performance of the multi-way stream buffer on our benchmark set. As expected, the performance on the instruction stream remains virtually unchanged. This means that the simpler single stream buffer will suffice for instruction streams. The multi-way stream buffer does significantly improve the performance on the data side, however. Overall, the multi-way stream buffer can remove 43% of the misses for the six programs, almost twice the performance of the single stream buffer. Although the matrix operations of *liver* experience the greatest improvement (it changes from 7% to 60% reduction), all of the programs benefit to some extent. Note also that *liver* makes unit stride accesses to its data structures.

4.2.3. Quasi-Sequential Stream Buffers

In the previous section only one address comparator was provided for the stream buffer. This means that even if the requested line was in the stream buffer, but not in the first location with the comparator, the stream buffer will miss on the reference and its contents will be flushed. One obvious improvement to this scheme is to place a comparator at each location in the stream buffer. Then if a cache line is skipped in a quasi-sequential reference pattern, the stream buffer will still be able to supply the cache line if it has already been fetched.

Figure 25 shows the performance of a stream buffer with three comparators. The quasi-stream buffer is able to remove 76% of the instruction-cache misses, an improvement of 4% over a

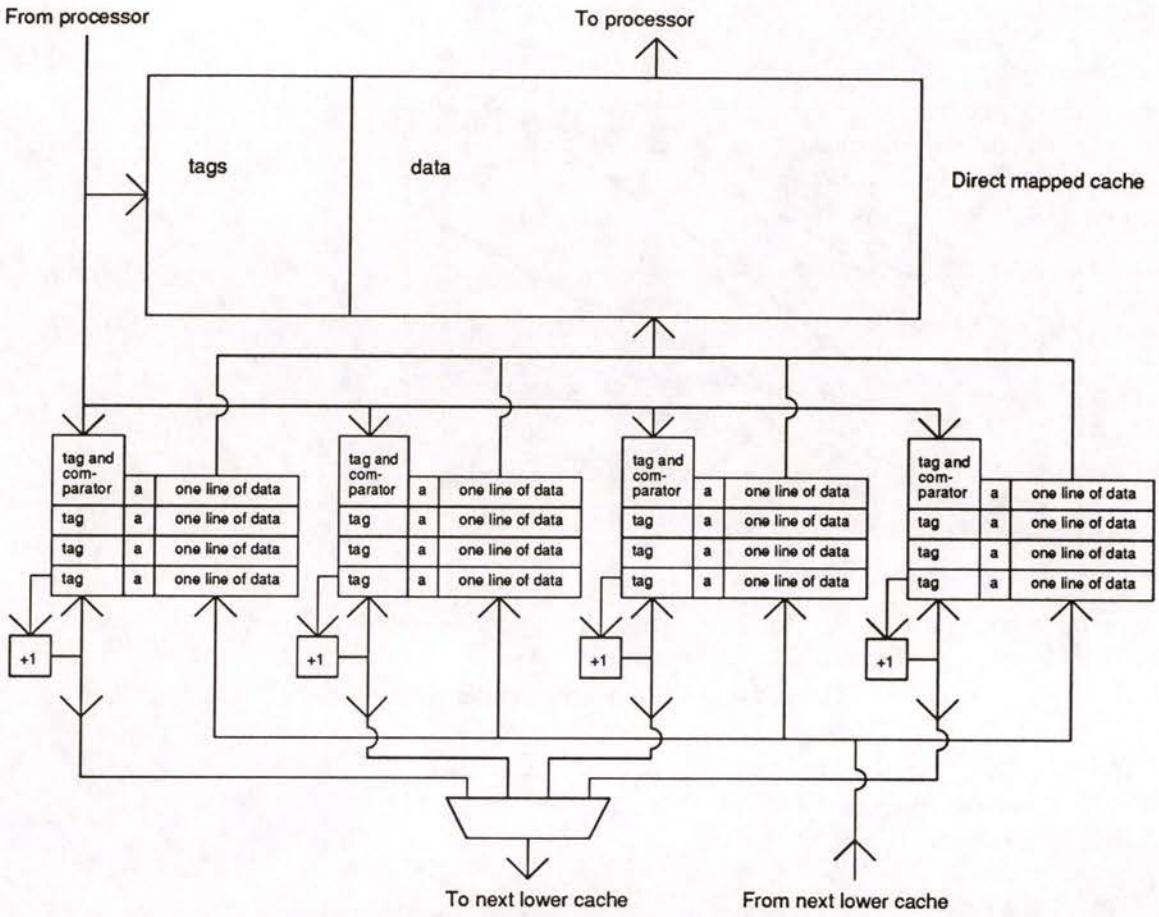


Figure 23: Four-way stream buffer design

purely sequential stream buffer, giving a 14% reduction in the number of misses remaining. This is probably due to the quasi-stream buffer's ability to continue useful fetching when code is skipped, such as when *then* or *else* clauses are skipped in *if* statements. The version simulated had three comparators, so it could skip at most 2 cache lines plus up to 3/4 of a cache line on either side depending on alignment, for a total of 16 to 22 instructions maximum. This compares with only 0 to 6 instructions that may be skipped in a sequential stream buffer (depending on branch alignment) without causing the stream buffer to be flushed.

The extra comparators of a quasi-stream buffer also improve the performance of a four-way data stream buffer. Overall, the four-way quasi-stream buffer can remove 47% of all misses, which is 4% more than the purely sequential four-way stream buffer.

Since the amount of hardware required for a few extra comparators on a single stream buffer is small, quasi-stream buffers seem like a useful generalization of sequential stream buffers for instruction streams. This is because only three additional comparators would be required to convert a four-entry sequential stream buffer into a quasi-stream buffer. However it may not be worthwhile for multi-way data quasi-stream buffers, since the number of extra comparators re-

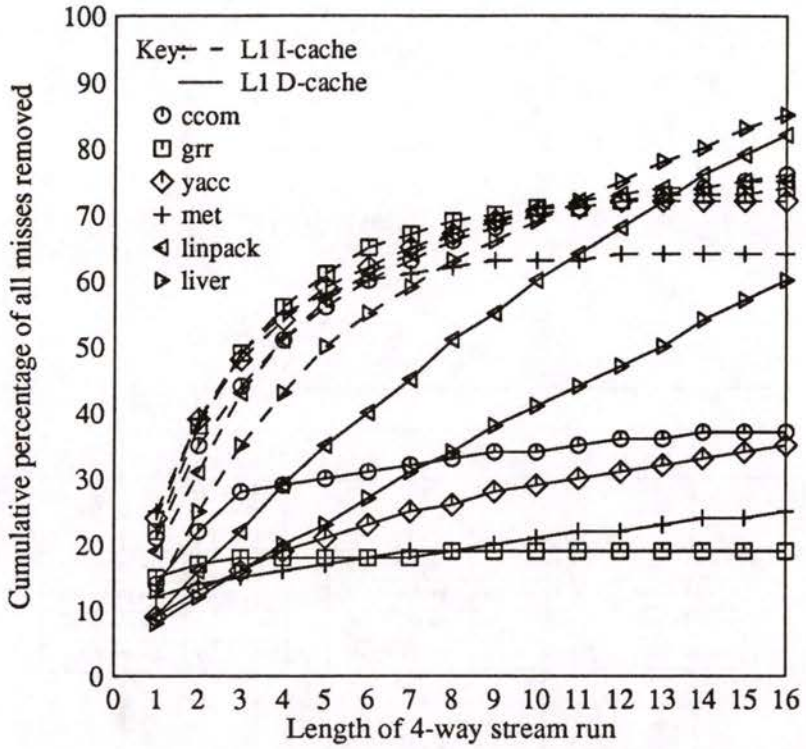


Figure 24: Four-way stream buffer performance

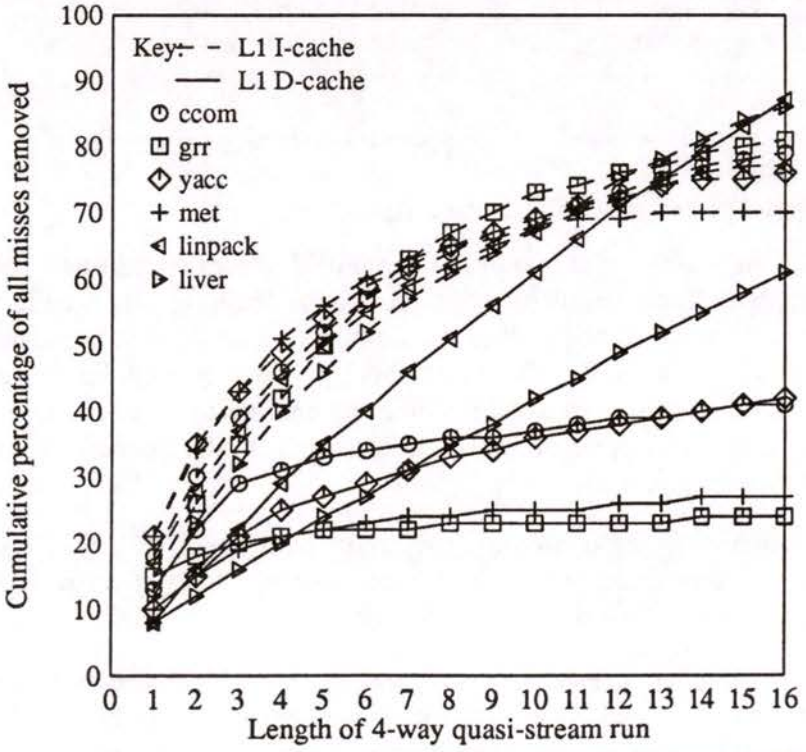


Figure 25: Quasi-sequential 4-way stream buffer performance

quired would be many times as large. An interesting area for software research is the ability of

compilers to reorganize code and data layouts to maximize the use of stream buffers. If techniques to optimize sequentiality of references are successful, the need for extra comparators on stream buffers will be lessened.

4.2.4. Stream Buffer Performance vs. Cache Size

Figure 26 gives the performance of single and 4-way stream buffers with 16B lines as a function of cache size. The instruction stream buffers have remarkably constant performance over a wide range of cache sizes. The data stream buffer performance generally improves as the cache size increases. This is especially true for the single stream buffer, whose performance increases from a 15% reduction in misses for a data cache size of 1KB to a 35% reduction in misses for a data cache size of 128KB. This is probably because as the cache size increases, it can contain data for reference patterns that access several sets of data, or at least all but one of the sets. What misses that remain are more likely to consist of very long single sequential streams. For example, as the cache size increases the percentage of compulsory misses increase, and these are more likely to be sequential in nature than data conflict or capacity misses.

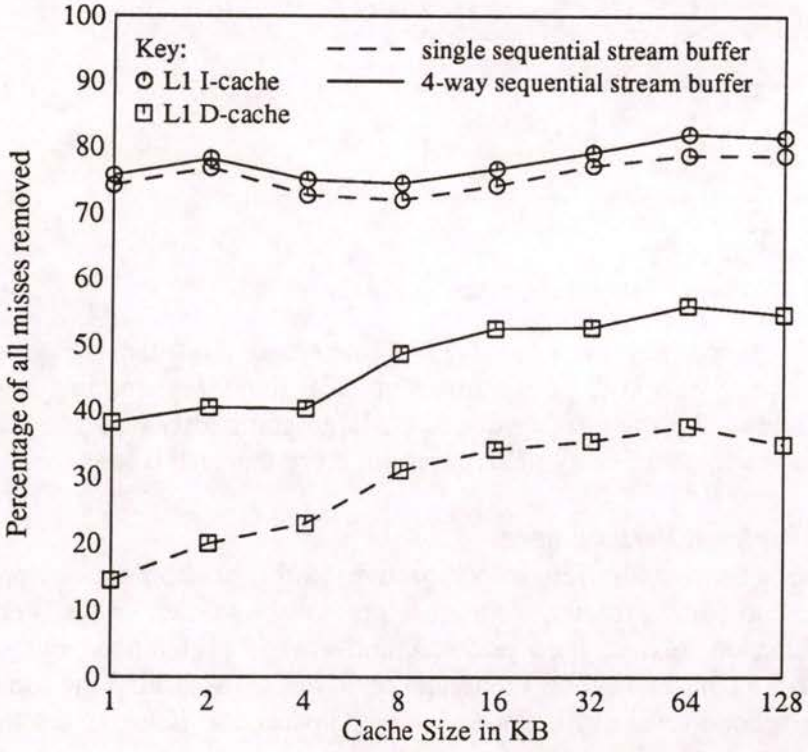


Figure 26: Stream buffer performance vs. cache size

4.2.5. Stream Buffer Performance vs. Line Size

Figure 27 gives the performance of single and 4-way stream buffers as a function of the line size in the stream buffer and 4KB cache. The reduction in misses provided by a single data stream buffer falls by a factor of 6.8 going from a line size of 8B to a line size of 128B, while a 4-way stream buffer's contribution falls by a factor of 4.5. This is not too surprising since data references are often fairly widely distributed. In other words if a piece of data is accessed, the odds that another piece of data 128B away will be needed soon are fairly low. The single data

stream buffer performance is especially hard hit compared to the multi-way stream buffer because of the increase in conflict misses at large line sizes.

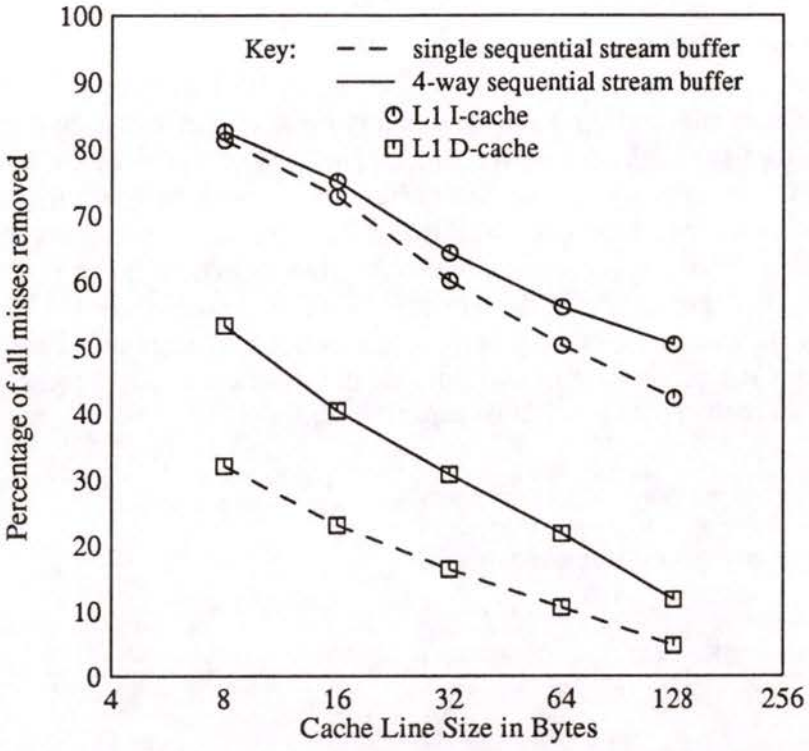


Figure 27: Stream buffer performance vs. line size

The instruction stream buffers perform well even out to 128B line sizes. Both the 4-way and the single stream buffer still remove at least 40% of the misses at 128B line sizes, coming down from an 80% reduction with 8B lines. This is probably due to the large granularity of conflicting instruction reference streams, and the fact that many procedures are more than 128B long.

4.2.6. Comparison to Classical Prefetch Performance

In order to put the performance of stream buffers in perspective, in this section we compare the performance of stream buffers to some prefetch techniques previously studied in the literature. The performance of prefetch on miss, tagged prefetch, and always prefetch on our six benchmarks is presented in Table 5. This data shows the reduction in misses assuming the use of these prefetch techniques with a second-level cache latency of one instruction-issue. Note that this is quite unrealistic since one-instruction issue latency may be less than a machine cycle, and second-level caches typically have a latency of many CPU cycles. Nevertheless, these figures give an upper bound of the performance of these prefetch techniques. The performance of the prefetch algorithms in this study is consistent with data earlier presented in the literature. In [14] reductions in miss rate for a PDP-11 trace on a 8KB mixed cache (only mixed caches were studied) with 16B lines and 8-way set associativity was found to be 27.8% for prefetch on miss, 50.2% for tagged prefetch, and 51.8% for prefetch always.

Table 6 compares the prefetch performance from Table 5 with the stream buffer performance presented earlier. On the instruction side, a simple single stream buffer outperforms prefetch on

fetch	ccom	yacc	met	grr	liver	linpack	avg
4KB instr. cache, direct-mapped, 16B lines, 1-instr prefetch latency:							
on miss	44.1	42.4	45.2	55.8	47.3	42.8	46.3
tagged	78.6	74.3	65.7	76.1	89.0	77.2	76.8
always	82.0	80.3	62.5	81.8	89.5	84.4	80.1
4KB data cache, direct-mapped, 16B lines, 1-instr prefetch latency:							
on miss	38.2	10.7	14.1	14.5	49.8	75.7	33.8
tagged	39.7	18.0	21.0	14.8	63.1	83.1	40.0
always	39.3	37.2	18.6	11.7	63.1	83.8	42.3

Table 5: Upper bound on prefetch performance: percent reduction in misses

miss by a wide margin. This is not surprising since for a purely sequential reference stream prefetch on miss will only reduce the number of misses by a factor of two. Both the simple single stream buffer and the quasi-stream buffer perform almost as well as tagged prefetch. As far as traffic is concerned, the stream buffer will fetch more after a miss than tagged prefetch, but it will not start fetching on a tag transition, so a comparison of traffic ratios would be interesting future research. The performance of the stream buffers on the instruction stream is slightly less than prefetch always. This is not surprising, since the performance of always prefetch approximates the percentage of instructions that are not taken branches, and is an upper bound on the reduction of instruction cache misses by sequential prefetching. However, the traffic ratio of the stream buffer approaches should be much closer to that of prefetch on miss or tagged prefetch than to prefetch always.

technique	misses eliminated
for 4KB direct-mapped instruction cache w/ 16B lines:	
prefetch on miss (1-instr latency)	46.3
single stream buffer	72.0
quasi-stream buffer (3 comparator)	76.0
tagged prefetch (1-instr latency)	76.8
always prefetch (1-instr latency)	80.1
for 4KB direct-mapped data cache w/ 16B lines:	
single stream buffer	25.0
prefetch on miss (1-instr latency)	33.8
tagged prefetch (1-instr latency)	40.0
always prefetch (1-instr latency)	42.3
4-way stream buffer	43.0
4-way quasi-stream buffer	47.0

Table 6: Upper bound of prefetch performance vs. stream buffer performance

Table 6 also compares the performance of stream buffers to other prefetch techniques for data references. Here both types of 4-way stream buffers outperform the other prefetch strategies. This is primarily because the prefetch strategies always put the prefetched item in the cache, even if it is not needed. The stream buffer approaches only move an item into to the cache if it is requested, resulting in less pollution than always placing the prefetched data in the cache. This

is especially important for data references since the spatial locality of data references is less than that of instruction references, and prefetched data is more likely to be pollution than are prefetched instructions.

Independent of the relative performance of stream buffers and ideal prefetch techniques, the stream buffer approaches are much more feasible to implement. This is because they can take advantage of pipelined memory systems (unlike prefetch on miss or tagged prefetch for sequential reference patterns). They also have lower latency requirements on prefetched data than the other prefetching techniques, since they can start fetching a block before the previous block is used. Finally, at least for instruction stream buffers, the extra hardware required by a stream buffer is often comparable to the additional tag storage required by tagged prefetch.

4.3. Combining Long Lines and Stream Buffers

Long cache lines and stream buffers can be used advantageously together, since the strengths and weaknesses of long lines and stream buffers are complimentary. For example, long lines fetch data that, even if not used immediately, will be around for later use. However, the other side of this advantage is that excessively long lines can pollute a cache. On the other hand, stream buffers do not unnecessarily pollute a cache since they only enter data when it is requested on a miss. However, at least one reference to successive data must be made relatively soon, otherwise it will pass out of the stream buffer without being used.

Table 7 gives the performance of various long-line and stream-buffer alternatives for a 4KB instruction cache. The first thing to notice is that all the stream buffer approaches, independent of their line size, outperform all of the longer line size approaches. In fact, the stream buffer approaches outperform a hypothetical machine with a line size that can be set to the best value for each benchmark. The fact that the stream buffers are doing better than this shows that they are actually providing an effective line size that varies on a per reference basis within each program. Also note that the line size used in the stream buffer approaches is not that significant, although it is very significant if a stream buffer is not used. Finally, the quasi-stream buffer capability approximates the performance of purely sequential stream buffers with longer line sizes. Consider for example a quasi-stream buffer than can skip two 16B lines. It will have a "prefetch reach" of between 16 and 22 four-byte instructions depending on alignment. This is a little longer span than a sequential 32B line stream buffer (8 to 15 instructions depending on alignment) and a little shorter than a sequential 64B line stream buffer (16 to 31 instructions). Thus it is not surprising that the performance of the 16B three-comparator quasi-stream buffer is between that of a 32B and a 64B line sequential stream buffer. Given that it is usually easier to make the cache line size equal to the transfer size, and that transfer sizes larger than 16B seem unlikely in the near future (at least for microprocessor-based machines), it seems that the use of quasi-sequential stream buffers with smaller line sizes such as 16B would be the most promising approach for the instruction cache. In particular if a quasi-sequential stream buffer is used, line sizes of greater than 32B have little benefit for 4KB instruction caches.

Table 8 gives the results for data stream buffers in comparison with longer line sizes, assuming there is no miss cache. Here the superiority of stream buffers over longer data cache line sizes is much more pronounced than with long instruction cache lines. For example, a four-way quasi-sequential data stream buffer can eliminate twice as many misses as the optimal line size per program, in comparison to only about 14% better performance for an instruction stream buff-

instr cache configuration (default does not include a miss cache)	misses eliminated
32B lines	38.0%
64B lines	55.4%
128B lines	69.7%
optimal line size per program	70.0%
16B lines w/ single stream buffer	72.0%
32B lines w/ single stream buffer	75.2%
16B lines w/ quasi-stream buffer	76.0%
64B lines w/ single stream buffer	77.6%
32B lines w/ quasi-stream buffer	80.0%
64B lines w/ quasi-stream buffer	80.2%

Table 7: Improvements relative to a 16B instruction line size without miss caching

er over an optimal per-program instruction cache line size. This is due to the wider range of localities present in data references. For example, some data reference patterns consist of references that are widely separated from previous data references (e.g., manipulation of complex linked data structures), while other reference patterns are sequential for long distances (e.g., unit stride array manipulation). Different instruction reference streams are quite similar by comparison. Thus it is not surprising that the ability of stream buffers to provide an effective line size that varies on a reference-by-reference basis is more important for data caches than for instruction caches.

data cache configuration (default does not include a miss cache)	misses eliminated
64B lines	0.5%
32B lines	1.0%
optimal line size per program	19.2%
16B lines w/ single stream buffer	25.0%
16B lines w/ 4-way stream buffer	43.0%
16B lines w/ 4-way quasi-stream buffer	47.0%

Table 8: Improvements relative to a 16B data line size without miss caching

Table 9 presents results assuming that longer data cache line sizes are used in conjunction with a four-entry miss cache. The addition of a miss cache improves the performance of the longer data cache line sizes, but they still underperform the stream buffers. This is still true even for a system with a different line size per program.

One obvious way to combine longer lines and stream buffers is to increase the line size up to the smallest line size that gives a minimum miss rate for some program. In our previous examples with a four-line miss cache this is a 32B line since this provides a minimum miss rate for *met*. Then stream buffers can be used to effectively provide what amounts to a variable line size extension. With 32B lines and a stream buffer a 68.6% further decrease in misses can be obtained. This does in fact yield the configuration with the best performance. Further increasing

data cache configuration (default includes 4-entry miss cache)	misses eliminated
32B lines	24.0%
16B lines w/ single stream buffer	25.0%
64B lines	31.0%
optimal line size per program	38.0%
16B lines w/ 4-way stream buffer	43.0%
16B lines w/ 4-way quasi-stream buffer	47.0%
64B lines w/ 4-way quasi-stream buffer	48.7%
32B lines w/ 4-way quasi-stream buffer	52.1%

Table 9: Improvements relative to a 16B data line size and 4-entry miss cache

the line size to 64B with a stream buffer is ineffective even though it reduces the average number of misses in configurations without a stream buffer. This is because the stream buffer will provide the same effect as longer cache lines for those references that need it, but will not have the extra conflict misses associated with longer cache line sizes.

5. Conclusions

Small miss caches (e.g., 2 to 5 entries) have been shown to be effective in reducing data cache conflict misses for direct-mapped caches in range of 1K to 8K bytes. They effectively remove tight conflicts where misses alternate between several addresses that map to the same line in the cache. Miss caches are increasingly beneficial as line sizes increase and the percentage of conflict misses increases. In general it appears that as the percentage of conflict misses increases, the percent of these misses removable by a miss cache also increases, resulting in an even steeper slope for the performance improvement possible by using miss caches.

Victim caches are an improvement to miss caching that saves the victim of the cache miss instead of the target in a small associative cache. Victim caches are even more effective at removing conflict misses than miss caches.

Stream buffers prefetch cache lines after a missed cache line. They store the line until it is requested by a cache miss (if ever) to avoid unnecessary pollution of the cache. They are particularly useful at reducing the number of capacity and compulsory misses. They can take full advantage of the memory bandwidth available in pipelined memory systems for sequential references, unlike previously discussed prefetch techniques such as tagged prefetch or prefetch on miss. Stream buffers can also tolerate longer memory system latencies since they prefetch data much in advance of other prefetch techniques (even prefetch always). Stream buffers can also compensate for instruction conflict misses, since these tend to be relatively sequential in nature as well.

Multi-way stream buffers are a set of stream buffers that can prefetch down several streams concurrently. In this study the starting prefetch address is replaced over all stream buffers in LRU order. Multi-way stream buffers are useful for data references that contain interleaved accesses to several different large data structures, such as in array operations. However, since the prefetching is of sequential lines, only unit stride or near unit stride (2 or 3) access patterns benefit.

The performance improvements due to victim caches and due to stream buffers are relatively orthogonal for data references. Victim caches work well where references alternate between two locations that map to the same line in the cache. They do not prefetch data but only do a better job of keeping data fetched available for use. Stream buffers, however, achieve performance improvements by prefetching data. They do not remove conflict misses unless the conflicts are widely spaced in time, and the cache miss reference stream consists of many sequential accesses. These are precisely the conflict misses not handled well by a victim cache due to its relatively small capacity. Over the set of six benchmarks, on average only 2.5% of 4KB direct-mapped data cache misses that hit in a four-entry victim cache also hit in a four-way stream buffer for *ccom*, *met*, *yacc*, *grr*, and *liver*. In contrast, *linpack*, due to its sequential data access patterns, has 50% of the hits in the victim cache also hit in a four-way stream buffer. However only 4% of *linpack*'s cache misses hit in the victim cache (it benefits least from victim caching among the six benchmarks), so this is still not a significant amount of overlap between stream buffers and victim caching.

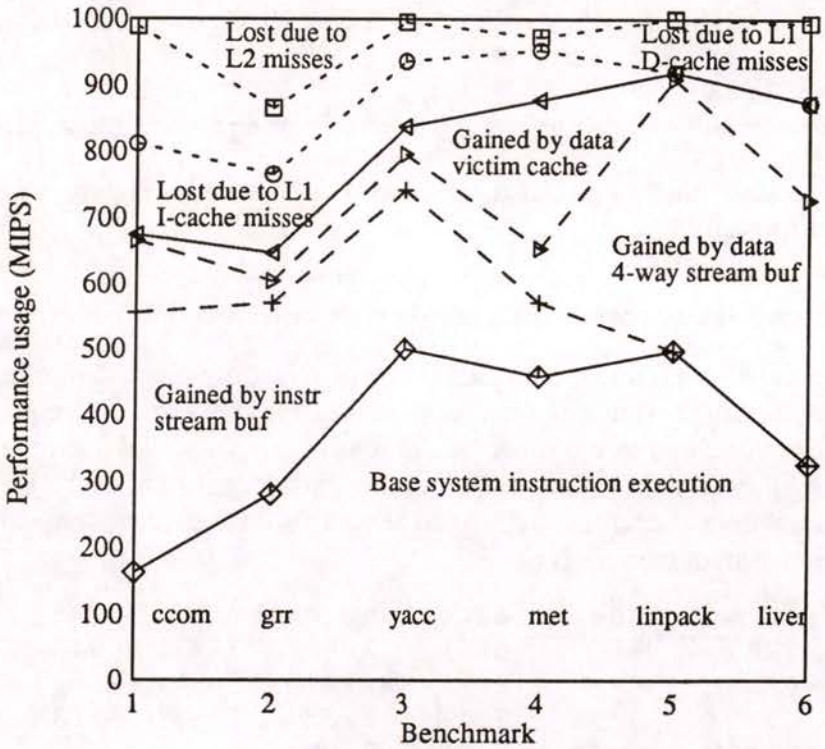


Figure 28: System performance with victim cache and stream buffers

Figure 28 shows the performance of the base system with the addition of a four entry data victim cache, a instruction stream buffer, and a four-way data stream buffer. (The base system has on-chip 4KB instruction and 4KB data caches with 24 cycle miss penalties and 16B lines to a three-stage pipelined second-level 1MB cache with 128B lines and 320 cycle miss penalty.) The lower solid line in Figure 28 gives the performance of the original base system without the victim caches or buffers while the upper solid line gives the performance with buffers and victim caches. The combination of these techniques reduces the first-level miss rate to less than half of that of the baseline system, resulting in an average of 143% improvement in system performance for the six benchmarks. These results show that the addition of a small amount of hardware can dramatically reduce cache miss rates and improve system performance.

One way of looking at the performance of victim caching and stream buffers is to consider the effective increase in cache size provided by these techniques. Table 10 gives the increase in cache size required to give the same instruction miss rate as a smaller cache plus a stream buffer. It is possible that by adding a stream buffer the compulsory misses are reduced to an extent that reduces the overall miss rate to a rate lower than that achieved by any cache with a 16B line size. Asterisks in Table 10 denotes situations where this occurs, or at least the miss rate is reduced beyond that of a 128KB cache, the largest size simulated. *ccom* has a particularly bad instruction cache miss rate, and it has a very large working set, so it benefits the most from instruction stream buffering.

program name	multiple increase in effective cache size 1K	2K	4K	8K	16K	32K	64K
ccom	26.3X	16.1X	7.0X	6.1X	4.1X	3.5X	*
grr	6.0X	3.5X	4.3X	3.4X	1.8X	2.7X	1.7X
yacc	7.5X	4.1X	3.0X	2.8X	1.9X	1.7X	*
met	3.2X	1.8X	2.1X	2.9X	1.9X	3.0X	1.9X
linpack	1.7X	1.9X	3.6X	*	*	*	*
liver	4.0X	2.0X	*	*	*	*	*

* denotes no cache size below 256KB attains as low a miss rate as cache with streambuffer

Table 10: Effective increase in instruction cache size provided by streambuffer with 16B lines

Corresponding equivalent increases in effective data cache size provided by the addition of a 4-entry victim cache and a 4-way stream buffer are given in Table 11. *linpack* and *liver* sequentially access very large arrays from one end to the other before returning. Thus they have very large effective cache size increases since with stream buffering they have equivalent cache sizes equal to their array sizes. (This assumes the stream buffer can keep up with their data consumption, which is true for our baseline system parameters.)

program name	multiple increase in effective cache size 1K	2K	4K	8K	16K	32K	64K
ccom	6.3X	5.0X	3.9X	3.1X	2.3X	1.8X	1.8X
grr	1.6X	1.5X	1.4X	1.2X	3.8X	*	*
yacc	1.6X	2.5X	1.7X	1.6X	1.7X	2.1X	*
met	1.4X	3.3X	1.2X	1.6X	3.3X	1.8X	*
linpack	98.3X	53.6X	30.4X	15.8X	*	*	*
liver	26.0X	16.0X	9.5X	8.4X	6.3X	3.4X	1.9X

* denotes no cache size below 256KB attains as low a miss rate as cache with 4-way streambuffer and 4-entry victim cache

Table 11: Effective increase in data cache size provided with stream buffers and victim caches using 16B lines

This study has concentrated on applying victim caches and stream buffers to first-level caches. An interesting area for future work is the application of these techniques to second-level caches. Also, the numeric programs used in this study used unit stride access patterns. Numeric programs with non-unit stride and mixed stride access patterns also need to be simulated. Finally, the performance of victim caching and stream buffers needs to be investigated for operating system execution and for multiprogramming workloads.

Acknowledgements

Mary Jo Doherty, John Ousterhout, Jeremy Dion, Anita Borg, and Richard Swan provided many helpful comments on an early draft of this paper. Alan Eustace suggested victim caching as an improvement to miss caching.

References

[1] Baer, Jean-Loup, and Wang, Wenn-Hann.
On the Inclusion Properties for Multi-Level Cache Hierarchies.
In *The 15th Annual Symposium on Computer Architecture*, pages 73-80. IEEE Computer Society Press, June, 1988.

[2] Borg, Anita, Kessler, Rick E., Lazana, Georgia, and Wall, David W.
Long Address Traces from RISC Machines: Generation and Analysis.
Technical Report 89/14, Digital Equipment Corporation Western Research Laboratory, September, 1989.

[3] Digital Equipment Corporation, Inc.
VAX Hardware Handbook, volume 1 - 1984.
Maynard, Massachusetts, 1984.

[4] Emer, Joel S., and Clark, Douglas W.
A Characterization of Processor Performance in the VAX-11/780.
In *The 11th Annual Symposium on Computer Architecture*, pages 301-310. IEEE Computer Society Press, June, 1984.

[5] Eustace, Alan.
Private communication.
February, 1989.

[6] Farrens, Matthew K., and Pleszkun, Andrew R.
Improving Performance of Small On-Chip Instruction Caches .
In *The 16th Annual Symposium on Computer Architecture*, pages 234-241. IEEE Computer Society Press, May, 1989.

[7] Hill, Mark D.
Aspects of Cache Memory and Instruction Buffer Performance.
PhD thesis, University of California, Berkeley, 1987.

[8] International Business Machines, Inc.
IBM3033 Processor Complex, Theory of Operation/Diagrams Manual - Processor Storage Control Function (PSCF).
Poughkeepsie, N.Y., 1982.

[9] Jouppi, Norman P., and Wall, David W.
Available Instruction-Level Parallelism For Superpipelined and Superscalar Machines.
In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272-282. IEEE Computer Society Press, April, 1989.

[10] Jouppi, Norman P.
Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.
In *The 16th Annual Symposium on Computer Architecture*, pages 281-289. IEEE Computer Society Press, May, 1989.

[11] Nielsen, Michael J. K.
Titan System Manual.
Technical Report 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

[12] Ousterhout, John.
Why Aren't Operating Systems Getting Faster As Fast As Hardware?.
Technical Report Technote 11, Digital Equipment Corporation Western Research Laboratory, October, 1989.

[13] Smith, Alan J.
Sequential program prefetching in memory hierarchies.
IEEE Computer 11(12):7-21, December, 1978.

[14] Smith, Alan J.
Cache Memories.
Computing Surveys :473-530, September, 1982.

DECEMBER 1990

WRL Technical Note TN-18

Predicting Program Behavior Using Real or Estimated Profiles

David W. Wall

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word "`help`" in the Subject line; you will receive detailed instructions.

Predicting Program Behavior Using Real or Estimated Profiles

David W. Wall

December, 1990

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

There is a growing interest in optimizations that depend on or benefit from an execution profile that tells where time is spent. How well does a profile from one run describe the behavior of a different run, and how does this compare with the behavior predicted statically by examining the program itself? This paper defines two abstract measures of how well a profile predicts actual behavior. According to these measures, real profiles indeed do better than estimated profiles, usually. A perfect profile from an earlier run with the same data set, however, does better still, sometimes by a factor of two. Using such a profile is unrealistic, and can lead to inflated expectations of a profile-driven optimization.

1. Introduction

Many people have built or speculated on systems that use a run-time profile to guide code optimization. Applications include the selection of variables to promote to registers [7,8], placement of code sequences to improve cache behavior [3,6], and prediction of common control paths for optimizations across basic block boundaries [2,5].

When such work is presented, two questions are often asked but seldom adequately answered. How well does a profile from one run predict the behavior of another? And how well can you do with an estimated profile derived from static analysis of the program? It is important to answer these questions in general terms as well as specific. A profile from a different run may be very useful for one kind of optimization but nearly useless for another kind. The optimization may require identifying the specific program entities that are most used, or it may require only identifying some that are used a lot.

This paper describes a study of how well an estimated profile predicts real behavior, and how well a profile from one run predicts the behavior of another run.

2. Methodology.

The *pixie* tool from Mips [4] instruments an executable file with basic block counting; when the instrumented program is run, it produces a table telling how many times each basic block was executed. From this table, in combination with static information from the executable file, we derive four kinds of profiles. The first is the *basic block* profile, which is just the mapping from each basic block to its execution count. The second is the *procedure* profile, which maps each procedure to the number of times it is entered. The third is the *call* profile, which maps each distinct call site to the number of times it is executed. The last is the *global variable* profile, which maps each global variable to the number of times it is directly referenced.

If we don't have basic block counts from pixie, we can try to estimate them. We first divide the program into basic blocks, and connect them into procedures and flow graphs based on the branch structure.* We then identify the loops by computing the dominator relation and finding the back edges, edges each of whose tail dominates its head. A loop consists of the set of back edges leading to a single dominator, together with the edges that appear on any path from the dominator to the head of one of the back edges [1]. We also build a static call graph by finding all the direct calls in the

* The Mips code generation is stylized enough that we can recognize indirect jumps that represent case-statements, and can deduce what the possible successor blocks are.

program; this graph will not include calls through procedure variables.

Given this information, we considered four different ways of estimating basic blocks counts. The first is the *loop-only* estimate, in which a block's count is initially 1 and is multiplied by 3 for each loop that contains it; this ignores the effects of the call graph. The second is the *leaf-loop* estimate, in which the loop-only count is multiplied by 1024 if the block is contained in a leaf procedure, 512 if it is no more than one from a leaf procedure, and so on with powers of 2 up to 1. The third is the *call-loop* estimate, in which the loop-only count is multiplied by the static number of direct calls of the block's procedure. The fourth is the *call+1-loop* estimate, which is the loop-only count is multiplied by one more than the static number of direct calls of the block's procedure. The call+1-loop estimate is like the call-loop estimate, except that procedures that are called only indirectly will not be shut out altogether; unfortunately procedures that are never called are similarly readmitted.

An optimizer would use a profile by selecting the most frequent entries in it and doing something special to them: promoting them to registers, optimizing them extra hard, or whatever. The question is how well a candidate profile, real or estimated, predicts the behavior described by a reference profile. For this study we considered two abstract methods of evaluating a candidate profile.

The first method, *specific matching,* is to take the top n entries of the candidate profile and see how many of them are also in the top n entries of the reference profile. For instance, consider the procedure profiles in Figure 1. If we let $n = 8$, we see that the first 8 members of the candidate profile include 5 of the first 8 members of the reference profile. Thus the candidate profile gets a score of 5/8, or 0.625.

306068	full_row
242254	force_lower
190252	malloc
190250	free
126993	set_or
86450	setp_implies
71835	d1_order
60790	set_clear
•	• •

878373	cdist0
245657	d1_order
138058	force_lower
72374	setp_disjoint
48672	cdist01
47029	malloc
47027	free
36491	full_row
•	• •
19131	set_or
15065	set_clear
•	• •
4792	setp_implies
•	• •

Figure 1. Candidate profile (left) and reference profile.

The second method, *frequency matching,* is to take the top n entries of the candidate profile and look up their frequencies in the reference profile, and then compare the total to the total of the top n entries of the reference profile. For example, taking the profile in Figure 1 and again assuming $n = 8$, the total of the candidate's top 8 entries

as revealed by the reference profile is 553250, while the total of the reference profile's top 8 entries is 1513681. By this measure, then, the candidate profile gets a score of 553250/1513681, or 0.365. Note that specific matching is symmetric (we get the same score comparing A to B as comparing B to A), but frequency matching is asymmetric.

Applying this approach to all four kinds of profiles, for different values of *n*, should give us some notion of how well one profile might predict another. To apply this understanding more specifically, we also did some rough computations of the stability of the profiles when applied in two specific ways. One application is the promotion of global variables to registers. The other is intensive optimization of the most frequently called procedures.

We should note one important limitation of this approach. It does not address the stability of a profile over successive versions of the same program undergoing development. One would expect that some kinds of profiles, such as global variable use or procedure invocation, might be relatively stable even when the program is modified. One might argue that a program under development will not be run enough times to merit profile-based optimization, but it would still be interesting to know whether it would be feasible. A thorough study of this question may be in order, but is not considered here.

3. Programs and data used

Our test suite consists of eleven programs. Two of them, a text editor and a drawing editor, are interactive. Two are CAD tools used at WRL. Two are different C compiler front ends; one is recursive descent, the other yacc-based. Three of them are SPEC benchmarks. Figure 2 describes the complete test suite.

program	*description*
bisim	multi-level machine simulator
bitv	timing verifier
udraw	drawing editor
egrep	file searcher
sed	stream editor
Gosling emacs	text editor
yacc	parser generator
ccom	Titan C front end
gcc1	gnu C front end
eqntott	truth table generator
espresso	set operation benchmark

Figure 2. The eleven test programs.

Wherever possible, we gave the programs quite different input data, in the hopes of maximizing the differences in their behavior. We ran bisim three different ways: completely high-level simulation, high-level functional units with a transistor-level register file, and transistor-level functional units with a high-level register file. Bitv was run to verify a datapath, a register file, and a write buffer. The drawing editor was used to draw schematics and also a home landscape design. Egrep and sed were run

with both simple and complicated patterns, and with large and small inputs. Emacs was used to edit source files, English text files, and very long simulation configuration files. Yacc was used with a high-level language grammar, an intermediate language grammar, and a command grammar for a window manager. The two C compilers were both run with two source files written by humans and two source files generated by the C++ front end. The eqntott and espresso benchmarks from SPEC were run with inputs provided by SPEC.

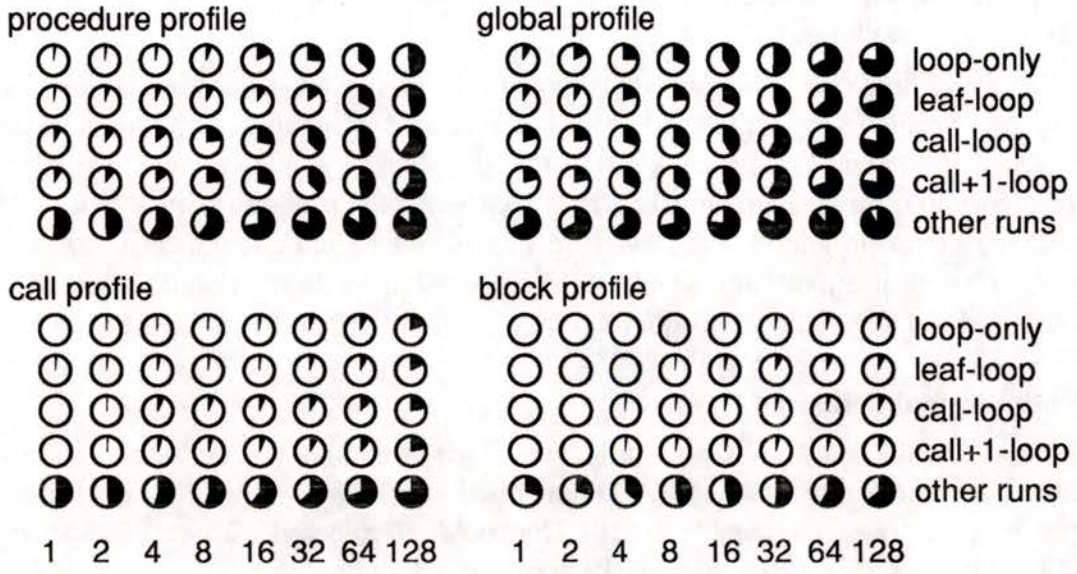


Figure 3. Average specific matching score.

4. Results

4.1. Specific matching

Our first result assumes that we score candidate profiles by the specific matching criterion, for n = 1, 2, 4, 8, 16, 32, 64, and 128. Given a test program and a value of n, we proceeded as follows. An estimated profile was scored against each real profile for the same test program; we then averaged these scores. Each real profile was scored against each of the *other* real profiles, but not against itself; we then averaged all the scores comparing two real profiles. For each test program and each value of n, this gave us 20 scores: the cross product of four profile classes and five estimate classes (more precisely, four estimate classes and also real profiles from other runs). We then averaged these scores over all programs; this double averaging gave each program equal weight even though some had more datasets than others.

The results are shown in Figure 3. The fraction of the circle filled with black is the score, so a completely black circle is perfect and a completely white circle is terrible. We can see that predicting which globals will be used is fairly easy, probably because there are fewer of them than there are of the other profiled entities. The call-loop estimates do rather better than the other estimates. As we would expect, actual

profiles do considerably better than estimates, but even actual profiles do disappointingly badly at predicting which basic blocks will be executed most.

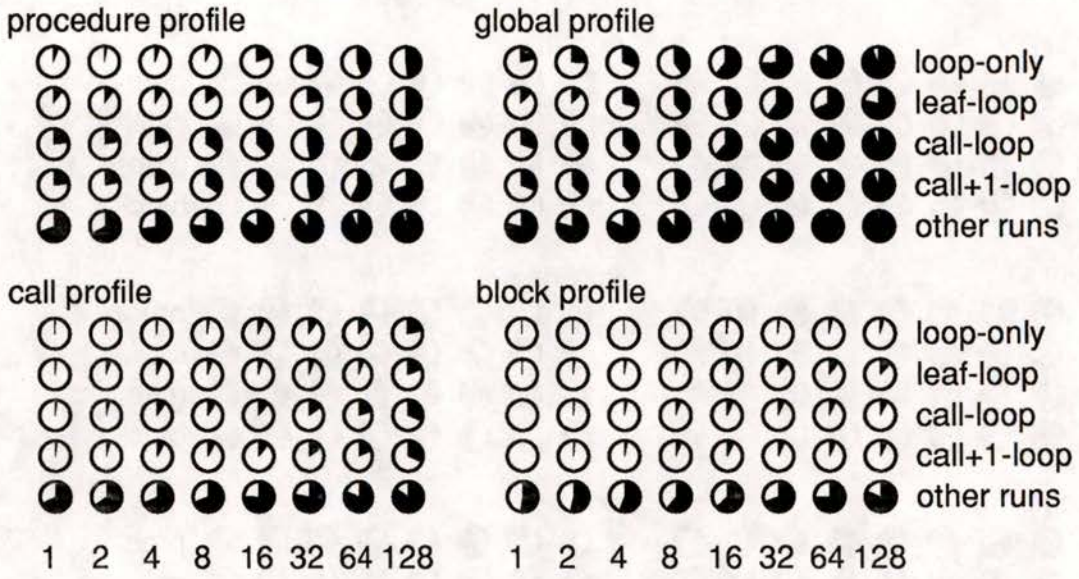


Figure 4. Average frequency matching score.

4.2. Frequency matching

Our next result has the same structure as the previous result, but it assumes frequency matching instead of specific matching. Again, we used n = 1, 2, 4, 8, 16, 32, 64, and 128. Each profile's scores were again averaged over all the profiles it was compared against, and the resulting averages were again averaged over the eleven test programs.

The results are shown in Figure 4. We were rather more successful at frequency matching than at specific matching.* The trends, however, are much the same: globals are easy to predict, blocks are hard, call-loop estimates work better than the others, and actual profiles work best of all.

4.3. Differences between test programs

There is a substantial variation in the predictability of the different programs. Figure 5 shows the average score for real (not estimated) profiles, using the frequency matching criterion. This is the fifth and tenth rows of Figure 4, broken down by program. Emacs is astonishingly predictable, perhaps because it is built around a Lisp interpreter, so that much of its control logic (and thus much of its variability) is hidden in the data structure. This argument would lead us to suppose that gcc1, with a table-driven parser, might be more predictable than ccom, with a recursive descent parser.

* This is not guaranteed in general: the candidate profile in Figure 1, for example, got a better score at specific matching.

But in fact ccom is noticeably more predictable than gcc1. The least predictable programs are sed and eqntott, which is a little surprising because they are among the smallest.

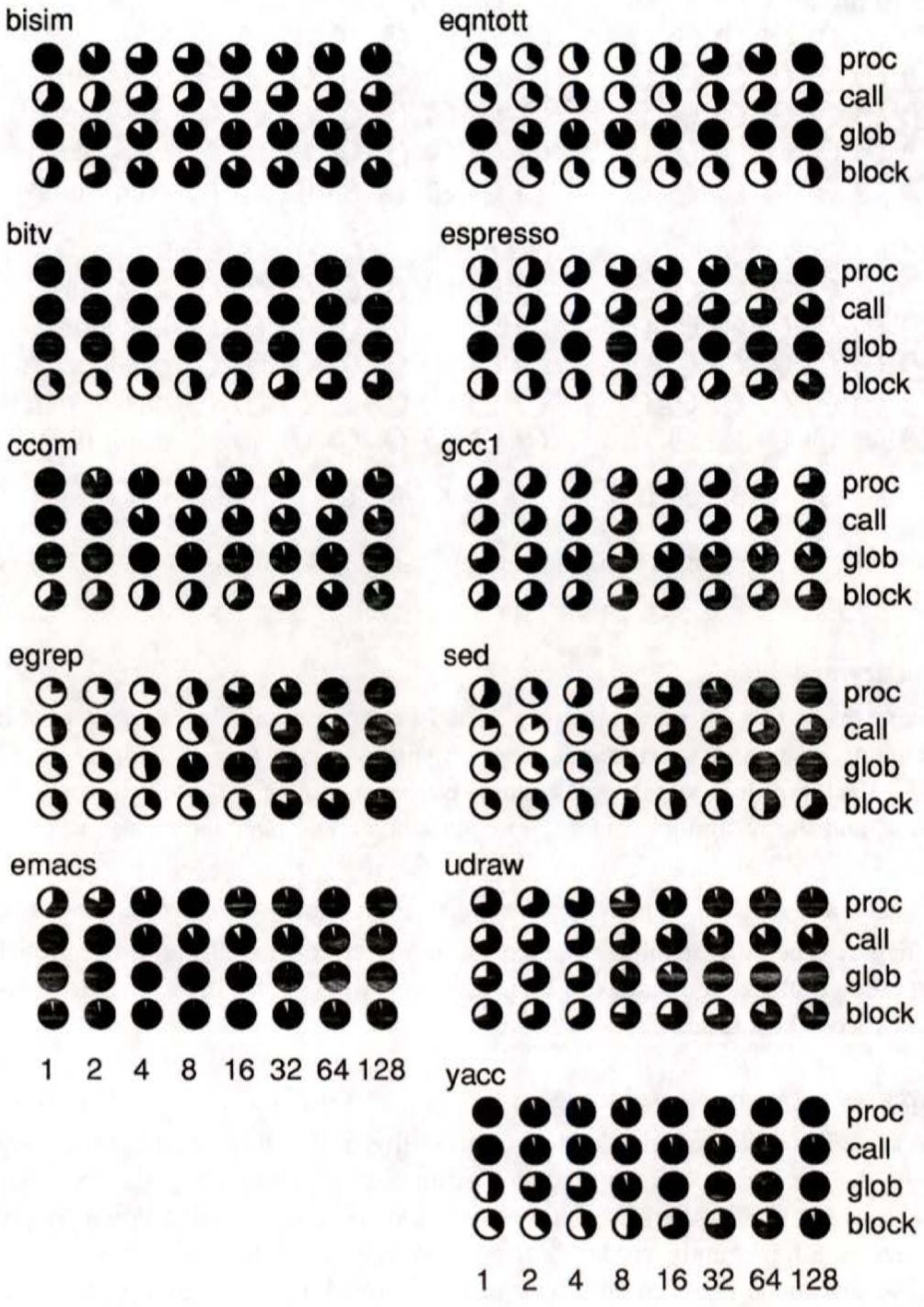


Figure 5. Average frequency matching scores for real profiles.

4.4. Global register allocation

To apply this technique to a realistic specific example, let us suppose that we suddenly have eight registers available that we can use to promote eight global variables or constants. They payoff of doing this is that all the loads and stores of the globals we select will be removed. We can estimate our improvement in performance by counting the executions of these loads and stores and dividing the total by the total number of instructions executed.* We did this both for a reference profile (to see how well we could possibly have done) and for a candidate profile, in each case computing the counts using the reference profile.

	improv	max	ratio
loop-only	1.3%	2.7%	
leaf-loop	1.1%	2.7%	
call-loop	1.2%	2.7%	
call+1-loop	1.3%	2.7%	
other runs	2.3%	2.7%	

Figure 6. Improvement from global register allocation.

The results are shown in Figure 6. This optimization by itself doesn't do a lot for performance: even if magically driven by the counts from the reference profile, the improvement in performance is only 2.7%. A good estimated profile gives us about half of the maximum possible performance improvement, and an actual profile gives us about 85% of the maximum.

4.5. Selective intensive optimization

As a second specific example, let us suppose we have an excellent but expensive optimization algorithm that will cut the execution time of any procedure in half, but that is so expensive that we can apply it only to 5% of our procedures. We will select as the procedures to optimize those we believe will be invoked most often, by picking the first 5% of the entries in the procedure invocation profile. As before, we will do this both for a candidate profile and also for a reference profile; we will compute the improvement in performance using only the counts from the reference profile.

The results are shown in Figure 7. This optimization would speed up our programs by a third if it were driven by a perfect profile. A real profile gives us about three-fourths of that, but even the best estimated profile -- which oddly enough was the simple loop-only estimate -- gives us barely one-fourth.

---

* This does not take pipeline stalls into account, nor does it consider cache effects, which are likely to increase the benefit of promoting globals to registers. It also assumes that the globals selected are not ineligible because of aliasing. We are interested only in rough numbers here, as an example.

	improv	max	ratio
loop-only	7.4%	31.2%	
leaf-loop	3.7%	31.2%	
call-loop	7.3%	31.2%	
call+1-loop	7.3%	31.2%	
other runs	24.2%	31.2%	

Figure 7. Improvement from selective intensive optimization.

5. Conclusions

Real profiles from different runs worked much better than the estimated profiles discussed in this paper. The best estimations were usually those that combined loop nesting level with static call counts. Basing the estimate on the procedure's distance from leaves of the call graph was less effective. There may of course still be better ways to estimate a profile: this is an interesting open question both in the general case and in specific applications.

Even a real profile was never as good as a perfect profile from the same run being measured. It was often quite close, however, and was usually at least half as good. Profile-based optimization would seem to have a future, but we must be careful how we measure it, lest we expect more than it can really deliver.

Acknowledgements

My thanks to Alan Eustace for goading me into finally doing this study, and to Patrick Boyle, Mary Jo Doherty, Ramsey Haddad, and Joel McCormack for helping me obtain some of the data.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools,* pp. 602-605. Addison-Wesley, 1986.

[2] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction,* pp. 37-47. Published as *SIGPLAN Notices 19* (6), June 1984.

[3] Scott McFarling. Program optimization for instruction caches. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 183-191, April 1989. Published as *Computer Architecture News 17* (2), *Operating Systems Review 23* (special issue), *SIGPLAN Notices 24* (special issue).

[4] MIPS Computer Systems, Inc. *Language Programmer's Guide,* 1986.

[5] Scott McFarling and John Hennessy. Reducing the cost of branches. *Proceedings of the 13th Annual Symposium on Computer Architecture,* pp. 396-403. Published as *Computer Architecture News 14* (2), June 1986.

[6] Karl Pettis and Robert C Hansen. Profile guided code positioning. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation,* pp. 16-27. Published as *SIGPLAN Notices 25* (6), June 1990.

[7] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation,* pp. 28-39. Published as *SIGPLAN Notices 25* (6), June 1990.

[8] David W. Wall. Global register allocation at link-time. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* pp. 264-275. Published as *SIGPLAN Notices 21* (7), July 1986. Also available as WRL Research Report 86/3.

WRL Research Reports

"Titan System Manual."
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

"Global Register Allocation at Link Time."
David W. Wall.
WRL Research Report 86/3, October 1986.

"Optimal Finned Heat Sinks."
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

"The Mahler Experience: Using an Intermediate Language as the Machine Description."
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

"The Packet Filter: An Efficient Mechanism for User-level Network Code."
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

"Fragmentation Considered Harmful."
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

"Cache Coherence in Distributed Systems."
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

"Register Windows vs. Register Allocation."
David W. Wall.
WRL Research Report 87/5, December 1987.

"Editing Graphical Objects Using Procedural Representations."
Paul J. Asente.
WRL Research Report 87/6, November 1987.

"The USENET Cookbook: an Experiment in Electronic Publication."
Brian K. Reid.
WRL Research Report 87/7, December 1987.

"MultiTitan: Four Architecture Papers."
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

"Fast Printed Circuit Board Routing."
Jeremy Dion.
WRL Research Report 88/1, March 1988.

"Compacting Garbage Collection with Ambiguous Roots."
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

"The Experimental Literature of The Internet: An Annotated Bibliography."
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

"Measured Capacity of an Ethernet: Myths and Reality."
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

"Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description."
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

"SCHEME->C A Portable Scheme-to-C Compiler."
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

"Optimal Group Distribution in Carry-Skip Adders."
Silvio Turrini.
WRL Research Report 89/2, February 1989.

"Precise Robotic Paste Dot Dispensing."
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

"Simple and Flexible Datagram Access Controls for Unix-based Gateways."
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

"Spritely NFS: Implementation and Performance of Cache-Consistency Protocols."
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

"Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines."
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

"A Unified Vector/Scalar Floating-Point Architecture."
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

"Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU."
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

"Integration and Packaging Plateaus of Processor Performance."
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

"A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance."
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

"The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance."
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

"Long Address Traces from RISC Machines: Generation and Analysis."
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

"Link-Time Code Modification."
David W. Wall.
WRL Research Report 89/17, September 1989.

"Noise Issues in the ECL Circuit Family."
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

"Efficient Generation of Test Patterns Using Boolean Satisfiablilty."
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

"Two Papers on Test Pattern Generation."
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

"Virtual Memory vs. The File System."
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

"Efficient Use of Workstations for Passive Monitoring of Local Area Networks."
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

"A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units."
John S. Fitch.
WRL Research Report 90/6, July 1990.

"1990 DECWRL/Livermore Magic Release."
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

"Pool Boiling Enhancement Techniques for Water at Low Pressure."
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

WRL Technical Notes

"TCP/IP PrintServer: Print Server Protocol."
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

"TCP/IP PrintServer: Server Architecture and Implementation."
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

"Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer."
Joel McCormack.
WRL Technical Note TN-9, September 1989.

"Why Aren't Operating Systems Getting Faster As Fast As Hardware?"
John Ousterhout.
WRL Technical Note TN-11, October 1989.

"Mostly-Copying Garbage Collection Picks Up Generations and C++."
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

"Limits of Instruction-Level Parallelism."
David W. Wall.
WRL Technical Note TN-15, December 1990.

"The Effect of Context Switches on Cache Performance."
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

"MTOOL: A Method For Detecting Memory Bottlenecks."
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

"Predicting Program Behavior Using Real or Estimated Profiles."
David W. Wall.
WRL Technical Note TN-18, December 1990.

digital™

INTEROFFICE MEMORANDUM

To: Distribution List
cc:

Date: 23-Jan-1992
From: Matt Reilly
Dept.: DCSS
Phone: (dtn)226-6633
Loc: LTN1-2/G10
E-Mail: GAZERS::REILLY

Subject: On Predicting System Performance for Large Workloads

---

The Problem:

Often we have a need to predict the performance of a new or proposed computer system against very large workloads. Unfortunately, many of these workloads are so large that running them in a simulator is impractical. For new systems, the logistics of setting up a large benchmark often prohibits such tests until very late in the debug phase.

Ideally we would be able to predict the performance of a new system given *real measurements of actual workloads* on a current (and presumably slower) system. In time we usually get a chance to collect a rather large body of information on our current system's behavior versus large workloads. This information can allow us to predict the performance of future systems without requiring extensive experimentation.

This note describes a technique to predict the speedup of a new system versus an older system for a large workload given:

- a scaling factor that relates the performance of the processor/cache components of the systems on a benchmark that is similar to the large application.
- a scaling factor that relates the memory request rates for the large application running on the two systems.
- the demand the application places on the older computer's memory system.
- the memory access time for the two systems.

All of these factors can be measured or estimated with available tools. The fundamental approach may be extended to include the effects of I/O system performance and contention for system resources.

The Premise:

It has proved useful to assume that while executing a compute-bound application, a computer is either thinking or waiting on a memory request. Thus the total run time for a given application can be expressed as the sum of this think time and the time spent waiting for memory:

$$T = T_{think} + T_{wait}$$

This is a rather old model[1] and has been extended by lots of folks to reflect the amount of time spent on operations within various levels of the memory hierarchy.

$$T = T_{think} + M_{c1}t_{c1} + M_{c2}t_{c2} + \cdots + M_{mem}t_{mem}$$

T_{wait} has been expanded into the sum of the cost of the memory accesses fielded by the first level cache, the second level cache, any intermediate caches, and the memory system. M_{cn} is the number of requests to cache n, and t_{cn} is the average time to satisfy each such request.

Now if we regroup the contributions to run time into those contributions that are determined by the processor and its caches and the contribution from the memory subsystem we get:

$$T = T_{proc} + M_{mem}\, t_{mem}$$

Where T_{proc} is the time spent executing code within the processor/cache ensemble. M_{mem} and t_{mem} are the memory request rate and the memory service time respectively.

Relating Two Systems:

Let's pick a particular reference machine (called **A**) and a large application. **A** takes T seconds to execute the application. Of these T seconds, T_{pA} are spent executing instructions and fetching things from **A**'s cache. During the application run, **A** waits for the memory to service M_A requests, thus $M_A\, t_{mA}$ seconds are spent waiting on **A**'s memory system. (The factor t_{mA} is the memory access time for system **A**.) Both T and M_A are easily measured on many of our current systems. (T can be measured with a stopwatch or various automated timers, M_A can be determined with a number of tools to be discussed later.) So now we can write:

$$T_A = T_{pA} + M_A\, t_{mA}$$

The object then is to find a way to relate **B**'s processing time (T_{pB}) to T_{pA} and to relate the amount of time **B** must wait for memory ($M_B t_{mB}$) to $M_A t_{mA}$. We do this with two linear scaling factors.

One could make the argument (and I do here) that for a large number of applications there exists some small benchmark that performs roughly the same kind of work and makes roughly the same use of the caches as the application without missing (and thus fetching from main memory) as often. A significant difference between the application and the benchmark is that the benchmark fits (for the most part) in the cache. A second and perhaps more significant difference between the two programs is that we can probably run this benchmark on a simulation model of the new system (call it **B**). If we run the benchmark program on system **A** and on the model of system **B**, measure the elapsed times and divide **B**'s elapsed time on the benchmark by **A**'s elapsed time, we get a **processor scaling factor** P_p that can relate the processing times T_{pA} and T_{pB} for any large application with a workload similar to that of the benchmark. We are now part of the way along to predicting **B**'s execution time for the large application:

$$T_B = T_{pB} + M_B\, t_{mB} = P_p\, T_{pA} + M_B\, t_{mB}$$

[1] I first saw this model in Strecker's 1972 thesis.

We can relate M_B to M_A by using trace driven simulation (driven by memory reference traces collected from the application running on system **A**) of **B**'s cache system. Assuming that **B**'s cache design is an improvement over that of **A**, system **B** will generate fewer misses than system **A**. The ratio of misses generated by **B** to the number generated by **A** is the **cache scaling factor** P_c. Now we have a complete expression for T_B in terms of what we already know or may calculate.

$$T_B = P_p T_{pA} + P_c M_A t_{mB} = P_p(T_A - M_A t_{mA}) + P_c M_A t_{mB}$$

where

T_A and T_B are the execution times for the application on systems **A** and **B**, respectively.

T_{pA} and T_{pB} are the processor/cache execution times for the application on **A** and **B**.

t_{mA} and t_{mB} are the memory access times.

M_A and M_B are the number of memory accesses generated by the application on **A** and **B**.

P_p is the processor scaling factor, calculated by running a small benchmark against **A** and **B**.

P_c is the cache scaling factor, calculated by address trace driven simulation of **B**'s cache.

The memory access times may be adjusted to account for contention in the interconnect or at the memories. Similar adjustments must be made if the processor-memory interconnect nears saturation. This must be calculated using M_A and the *writeback request rate* as the request rate for system **A**. (More about this later.)

Collecting the Factors:

The model described here requires calculation of the two scaling factors and a count of the number of memory references generated by system **A** for a given application. A number of tools exist that make these tasks possible.

P_p, the processor scaling factor, is simply the ratio of run times for a benchmark on the two systems. For the factor to be useful, the benchmark must fit in the caches of both systems, and perform work that is similar to the large application of interest. The choice of a benchmark might be validated by using a tool such as PerfMon (the NVAX internal monitoring hardware with the software written by John Brown and Rick Gillett) to compare the candidate benchmark to samples of the execution of the application of interest. The object is to verify that the work the benchmark does is representative of that done by the application. Showing a correspondence between the two on a VAX may even be sufficient to allow comparison between VAX systems and those built upon other instruction sets. In the absence of a tool like PerfMon, one may resort to Kentucky Windage, engineering judgment, or even simulation of the benchmark and parts of the application of interest. With time and experience of varied applications we should get better at estimating P_p.

P_c, the cache scaling factor, is a bit more complicated. In simpler times, when caches were small, we could have used popular rules of thumb or Simon Steely's ''Lots of Caches''. While the predictive value of these tools is clear, their general nature precludes using them to predict behavior of specific, large, and often pathological applications. There are a number of tools to

aid us in calculating P_c however. The most reliable approach is to run the application on a current system (**A**), collect a trace of all of the requests sent to the memory system, and then process this trace through a trace driven simulation of **B**'s cache. This will produce a new trace with (1 - P_c) fewer misses than found in the original trace of **A**'s activity. These traces may be collected using the Xpod[2] module with a Biomation logic analyzer, or soon, with the Xpod module and a Turbochannel Trace Collection Module (TTCM). The Biomation is capable of collecting traces of up to 2.5 million entries. The TTCM may collect traces as long as 50 million entries. There are other tools available for long address trace collection as well. Both tools are generally non-invasive.

M_A can be calculated from the trace used to generate P_c or by using PerfMon, an Xwatch[3], or any number of other non-invasive measurement devices.

An Example:

As an example, let's predict the elapsed time for a VAX 6610 to execute 750x750 LINPACK using what we know of the program's behavior on a 6510. John Shakshober has supplied SPECmark figures for both machines that indicates a value of $P_p = 0.33$ to be a reasonable estimate. (That is, the 6610 takes one third the time to execute a floating point benchmark that fits entirely in its cache compared to the time taken by a 6510.) Sean Reilly has also collected a memory reference trace for 750x750 LINPACK using an Xpod and a Biomation logic analyzer. Simon Steely's cache simulator suggests that the 6610's cache will generate 15% fewer memory accesses than a 6510 did *for the actual application*. So $P_c = 0.85$. The 6510 made about 172,000 read references per second[4] during the LINPACK run (which itself took 160 seconds). Thus $T_A = 160$ and $M_A = 27$ million. The memory access times for both the 6610 and 6510 are approximately 1 microsecond: $t_{mA} = t_{mB} = 1.0\text{x}10^{-6}$. Plugging all this into our expression for T_B we get:

$$T_B = 0.33\,(160 - 27 \times 10^6 \cdot 10^{-6}) + 0.85 \cdot 27 \times 10^6 \cdot 10^{-6} = 67\ seconds$$

This is very close to the actual measured result of 64 seconds.

I/O and Memory Contention:

For applications that are not compute-bound we need to modify our original model to include I/O wait time:

$$T = T_{think} + T_{wait_mem} + T_{wait_IO}$$

Fortunately, T_{wait_IO} can be found for most applications by a number of means. For VMS applications, SPM, VPA, PCA, DECtrace, or even MONITOR can produce reasonable estimates of the amount of time an application spends waiting for I/O completion. It will then be necessary to develop a scaling factor for I/O performance to relate the two systems.

[2]Xpod is a simple probe for the XMI designed by Dave Hartwell.

[3]Xwatch is a counter and monitor device for the XMI designed by Bimal Sareen.

[4]I assume here that the processor doesn't wait on writeback traffic. This assumption will require some testing.

Memory contention can be modeled with a simple queuing model for most cases. In any event, the problem has proved tractable. A closed form solution is unlikely however, as in the presence of contention t_{mB} is dependent on the **rate** of requests to the memory system (the ratio M_B / T_B) and T_B is dependent on t_{mB}. Further, one must also include the traffic due to cache write-back operations. This has been dealt with successfully in models developed for YUKON, TurboLaser, ScalP, and others.

Summary:

To review the necessary steps in predicting the performance of application X on a new system (call it **B**):

1. Run X on a current system (system **A**) and
   a. Measure the elapsed time for the application run. This is the value of T_A.
   b. Count the number of memory references made during the application run. This is M_A.
   c. Collect a representative trace of the memory references made during the run.
2. Run a benchmark whose ''profile'' is similar to the application on **A** and a simulation of **B**. Divide the execution time for **B** by the execution time for **A**. This is P_p.
3. Use the trace collected in step 1c to drive a cache simulator for system **B**. Use the simulation results to estimate the proportion of references from the trace that are eliminated by **B**'s cache. This proportion is $(1 - P_c)$. Note that one may just as easily calculate M_B directly from the simulation results. (Sean Reilly and Simon Steely have both implemented simulators that serve this purpose.)
4. Provided that the total traffic (M_x plus the write-back traffic) for both systems is well below the bus bandwidth limit, use the typical memory access times for each system as t_{mA} and t_{mB}. If either bus is near saturation, model or measure the contention and queuing effects.
5. Ensure that I/O performance is not a factor. If I/O wait time represents a significant proportion of T_A it may become a much more significant component of T_B. Additional modeling may be required to account for I/O performance in these cases.
6. Plug the values determined for P_p, P_c, T_A, M_A, t_{mA}, and t_{mB} into the equation to find T_B.

A Tool Catalogue:

This approach relies on our ability to collect large address traces from large applications. Deb Slater, Sean Reilly, Simon Steely and others have been collecting 2.5 million entry traces from 6000 series machines with an Xpod module and a logic analyzer. In a short time we should have the ability to collect longer traces using the same Xpod with the new TTCM (Turbochannel Trace Collection Module) and a DS5000. (The TTCM is already capable of collecting traces of software activity on a DS5000 using a semi-invasive technique that I have described elsewhere.)

The large traces allow us to calculate P_c the cache performance scaling factor. The processor performance scaling factor (P_p) can be calculated using the various processor performance models developed by Simon Steely and Eric Rasmussen, the EV5 model (from Mike Uhler,

Pete Bannon, Jim Keller, and Paul Rubenfeld), the EV4 model (from Dirk Meyer and company), or any of the processor simulators done for the various projects.

Validating the choice of a benchmark that matches the application of interest may be greatly aided with the PerfMon hardware performance monitor for the NVAX processor. PerfMon provides execution and cache performance information to a fine level of detail. In the absence of other insight, one may use PerfMon to confirm the choice of the benchmark used in calculating P_p.

Conclusions:

I believe that we are at (or at least near) the point where we can use the tools in existence and soon to be available to predict performance of and perhaps even steer development of future systems.

I need to do some more work at validating the model against a wider range of programs. We have had trouble with one of our test cases: 600x600 LINPACK generated traces of memory references from a 6510 that produced wildly different values for P_c when run against a cache simulator for the 6610. Estimates of P_c ranged from 0.45 to 0.65. Examination suggests that the cache traces were not representative samples of the application. Each trace was of limited length (approximately 2.5 million samples). The availability of the TTCM to collect much longer traces should shed a little more light on this problem.

In the coming weeks I'd like to try predicting single processor TP performance based on this technique. Generally, I believe the technique can be used to scale service rates for the TP server processes. We can use this figure as an input to a simple queuing model. The queuing model would allow prediction of the maximum TPS deliverable within the response time limit. Multiprocessor TP performance and the MP scaling issue will certainly take much more study, but this looks like a reasonable start..

I'd appreciate any comments you might have, or advice as to where this might be applied to good advantage.