WRITES CACHES AS AN ALTERNATIVE TO WRITE BUFFERS

Brian K. Bray
M. J. Flynn

Technical Report No. CSL-TR-91-470

April 1991

Supported by NASA under NAG2-248 using facilities supplied under NAGW 419.

WRITES CACHES AS AN ALTERNATIVE TO WRITE BUFFERS

by

Brian K. Bray

M. J. Flynn

Technical Report No. CSL-TR-91-470

April 1991

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

Write buffers help unbind one level of a memory hierarchy from the next, thus write buffers are used to reduce write stalls. Write buffers are used in write-through systems so that writes can occur at the rate the cache can handle them, but write buffers don't reduce the number of writes, or cluster writes for block transfers. A ***write cache*** is a cache that uses an allocate on write miss, write-back, no allocate on read miss strategy. A ***write cache*** tries to reduce the total number of writes (write traffic) to the next level by taking advantage of the temporal locality of writes. A ***write cache*** also groups writes for block transfers by taking advantage of the spatial locality of writes. We have found that small ***write caches can*** significantly reduce the write traffic to the first write-back level after the processor's register set. Systems that would benefit from reduced write traffic to the first write-back level would benefit from using a ***write cache*** instead of a write buffer. The temporal and spatial locality of writes is very important in determining what organization ***the write cache*** should have.

Key Words and Phrases: write-back, write-through, write buffer, write cache

Contents

1 Introduction

One important cache design decision is whether a cache should be write-through or write-back [Smi82]. A write-through cache has writes go to both the current level and the next level of the memory hierarchy. A write-back cache has writes go to the current level, and the data is only written to the next level when forced by line replacement or flushing.

Write-back caches have advantages over write-through caches: 1.) write-back caches use less memory bandwidth for writes; 2.) write backs occur in line size units which facilitate block mode transfers; and 3.) processor writes occur at the rate the cache can handle them, not at the rate the memory hierarchy can handle them.

Write-through caches have advantages over write-back caches: 1.) a simpler operating system kernel, since there is no need to determine when to flush dirty lines; 2.) no error-correcting circuitry is needed on the first-level cache, because if a parity error does occur, a cache miss can be generated and the data fetched from the next level; and 3.) stores can be made faster, resulting in less stalls due to cache contention [FKH87]. However, as the data read buffering increases, unbuffered write traffic becomes dominant (Figure 1). The amount of write traffic can become a limit to execution rate and can also reduce performance by increasing the busy time of the next level, thus increasing the read miss penalty (see Appendix A).

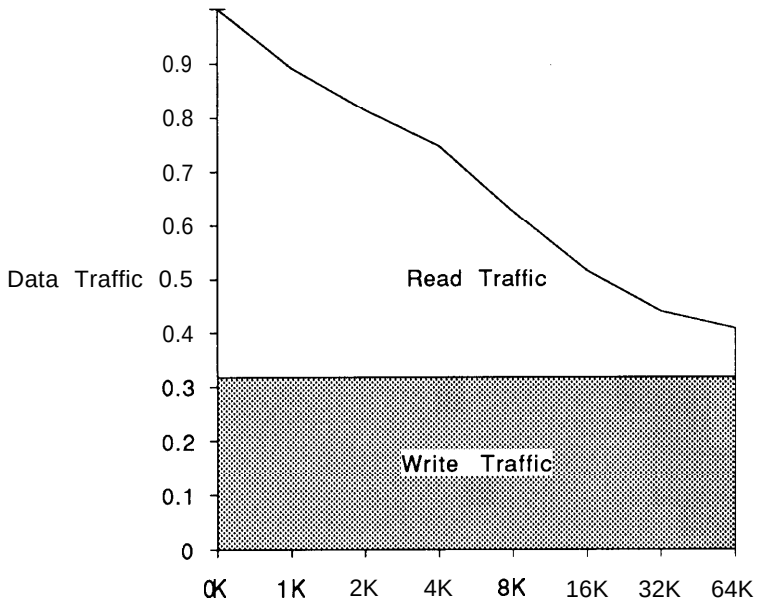

Figure 1: Unbuffered Write Traffic vs Buffered Read Traffic

Write buffers help unbind one level of a memory hierarchy from the next, thus write buffers are used to reduce write stalls. Write buffers are used in write-through systems so that writes can occur at the rate the cache can handle them, but write buffers don't reduce the number of writes, or cluster writes for block transfers. In [FKH87], the authors mention that they modified the write buffer (one entry, 16 byte length) to be a write-allocate cache, this was to reduce the write traffic from the write-through cache. We call this class of modified write buffers write ***caches.***

A write ***cache*** is a cache that uses an allocate on write miss, write-back, no allocate on read miss strategy, with a single combined dirty/valid bit per byte (byte subblocks with no fetching). A ***write cache*** tries

to reduce the total number of writes (write traffic) to the next level by taking advantage of the temporal locality of writes. A ***write cache*** also groups writes for block transfers by taking advantage of the spatial locality of writes.

2 Write Cache Performance Tradeoffs

A ***write cache*** is different from ordinary caches in that ordinary caches have been added principally to reduce the read access time of memory references (or miss penalty [Jou90]), while a ***write cache*** is used to reduce busy time of the next level, thus indirectly reducing miss penalty. A ***write cache*** differs from most caches in that it does not allocate on a read miss since the goal is to reduce write traffic. To reduce the write traffic, a ***write cache*** uses the same ideas that make regular caches work, temporal and spatial locality of references. But ordinary caches are based on the temporal and spatial locality of reads to reduce the read misses, while a ***write cache*** is based on the locality of writes.

Accessing patterns that sweep through a large memory space usually cause cache organizations to exhibit anomalous behavior. Therefore, the benchmarks (see Appendix B) are separated into two groups, depending on whether or not the benchmark is characterized by having data write references dominated by sweeping through large structures. We call the benchmark set which is characterized by having write references sweep through large data structures, ***the Sweep Benchmark*** Set. We call the other benchmark set ***the Non-Sweep Benchmark Set.***

Write caches have three causes for write traffic: compulsory, capacity, and conflict. The number of unique byte locations written during the execution of a program is the compulsory number of bytes to be written, but their spatial locality and the transfer size determines the compulsory write traffic. Capacity write traffic results when there are more unique byte locations which are dirty and will be written to in the future than there is capacity to buffer them. Conflict write traffic is due to the occurrence of unoptimal mapping, such as when an active write area is mapped to the same location in the ***write cache*** as another active write area even though there is currently an unactive area available.

Most of the control over the compulsory, capacity, and conflict write traffic is contained in the way the algorithm and the compiler require/use data structures. The benchmark architecture is the MIPS R2000/R3000 with compiler optimization. The register allocator does a relatively good job of reducing loads and stores, especially to the run-time stack. Usually there is a high degree of temporal and spatial locality in the references to the stack, so the results presented should be better for machines with fewer registers or poorer register allocation.

Area is usually limited, so performance of caching schemes is usually capacity limited. As a result, the architect is usually interested in organizations which reduce capacity misses and the miss penalty. Some of the most important parameters of caches are the number of lines, transfer size, line size, and associativity. In the next sections, we discuss how these parameters affect ***write caches.***

2.1 Transfer Size

Transfer size, whether achieved with bus width or block mode transfers, is important. Increasing the transfer size can reduce the miss penalty of capacity, conflict, and compulsory write traffic. (Write traffic is the number of write transfers.) However the effectiveness of a larger transfer size is directly proportional to the spatial locality that the write traffic exhibits. Increasing the maximum transfer size

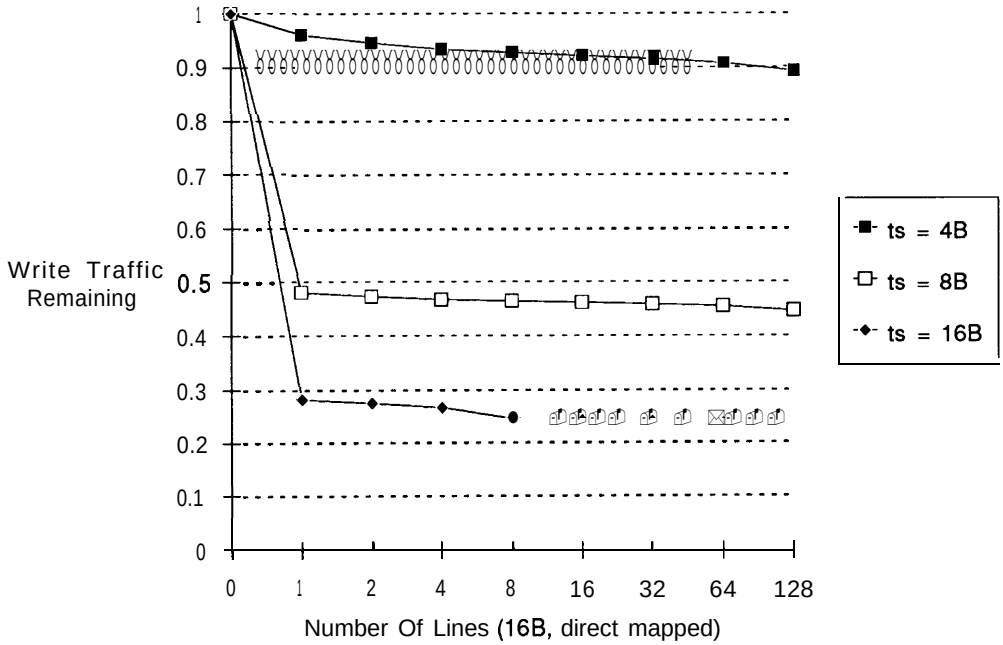

Figure 2: Effect Of Transfer Size On Write Traffic Reduction (Sweep Benchmark Set)

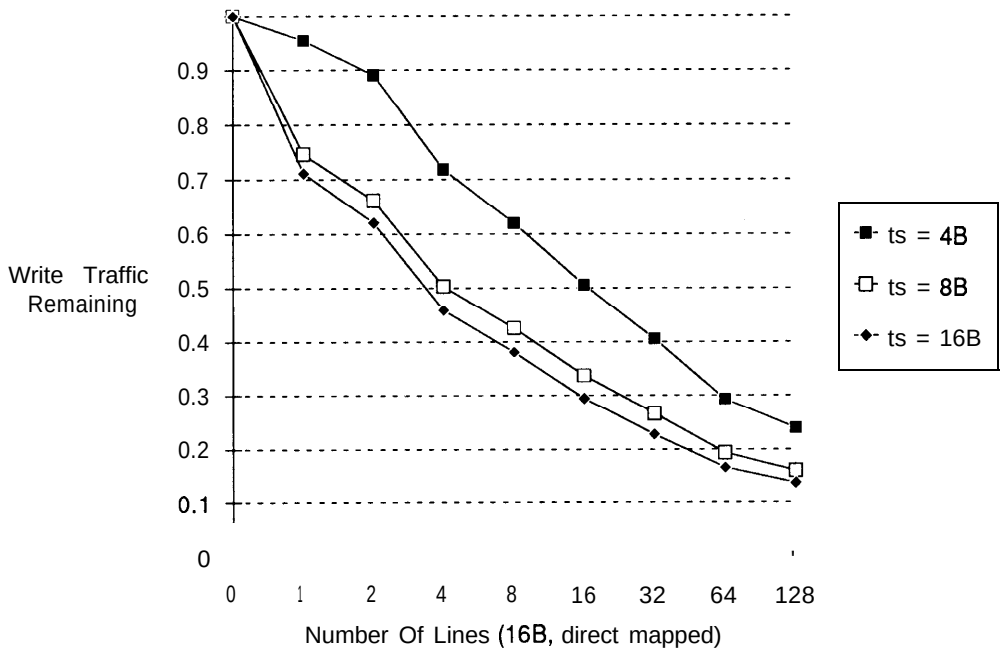

Figure 3: Effect Of Transfer Size On Write Traffic Reduction (Non-Sweep Benchmark Set)

(ts) for the sweep benchmark set (Figure 2) results in a significant decrease in the write traffic, thus the sweep benchmark set exhibits high spatial locality for writes. Also, note that increasing the ***write cache*** size beyond one line had little effect, thus the sweep benchmark showed low temporal locality for writes. Increasing the maximum transfer size (ts) for the non-sweep benchmark set (Figure 3) results in some write traffic reduction, but not as significant as that for the sweep benchmark set. The non-sweep benchmark set showed little locality, especially beyond 8 bytes; increasing the transfer size from 8 bytes to 16 bytes had little effect. Unlike the sweep benchmark set, the non-sweep benchmark set showed significant temporal locality for writes, as increasing the number of lines beyond one noticably decreased the write traffic.

As the number of lines in a ***write cache*** increases, the time an object remains in the ***write cache*** should also increase. The longer an object remains in ***the write cache,*** it would seem more likely a spatially adjacent object would also be written, thus increasing the spatial locality. The increase in spacial locality results in an increased importance of larger transfer size. If this were true, then the performance curves for different transfer sizes should diverge. In Figure 3, the transfer size curve of 4 bytes diverges from the 8 byte and 16 byte curves as the number of lines increase from 1 to 2, and the 16 byte transfer size curve very slightly diverges from the 8 byte transfer size curve, as the number of lines increase from 1 to 8. However, the transfer size curve of 4 bytes converges to the other curves as the number of lines increases above 2. The 16 byte transfer size curve approaches to the 8 byte transfer size curve as the number of lines increase above 8. The transfer line curves converge (thus decreasing the importance of a large transfer size), because the write misses are decreasing more than the spatial locality is increasing.

2.2 Transfer Scheme

In the previous discussion, only the valid/dirty bytes were written to the next level. A transfer size of 4 bytes would make 1 transfer if all the bytes were valid in the 4 byte unit, else it would transfer the bytes one at a time causing several transfers thus causing increased write traffic. The 8 byte trasfer size would transfer units in sizes of 1, 4 or 8 bytes per transfer, and the 16 bytes transfer size would transfer units of 1, 4, 8, or 16 bytes per transfer. So unless all the bytes in the transfer sized unit were valid/dirty when replaced, excess write traffic would occur. If a more complex scheme that does not require smaller than maximum transfer sized units can be used, then the write traffic should decrease. One way to do this is to always send the valid bits along with the maximum transfer sized units, and use the valid bits as byte enables (indicated as vbit in Figure 4). This more complex scheme made no write traffic reduction for the sweep benchmark set, and little difference for the non-sweep benchmark set (Figure 4), except for the 16 byte transfer size. However, as ***the write cache*** size increased, the number of misses decreased, so the importance of efficiently handling misses decreased.

2.3 Line Size

From the results with varying the transfer sizes and number of lines, we conclude that for the sweep benchmark set, the transfer size was important and the temporal locality wasn't (at least for the sizes of ***write caches*** we are looking at). Therefore, the effect of line size (ls) larger than the transfer size should be minimal, and it is (Figure 5). Therefore, the line size should be no larger than the transfer size.

For the non-sweep benchmark set, the number of lines (capturing temporal locality) was more important than transfer size, so decreasing the line size (ls) for a fixed number of lines causes a noticable increase in the write traffic (Figure 6). However, comparing the performance of different line sizes with equal

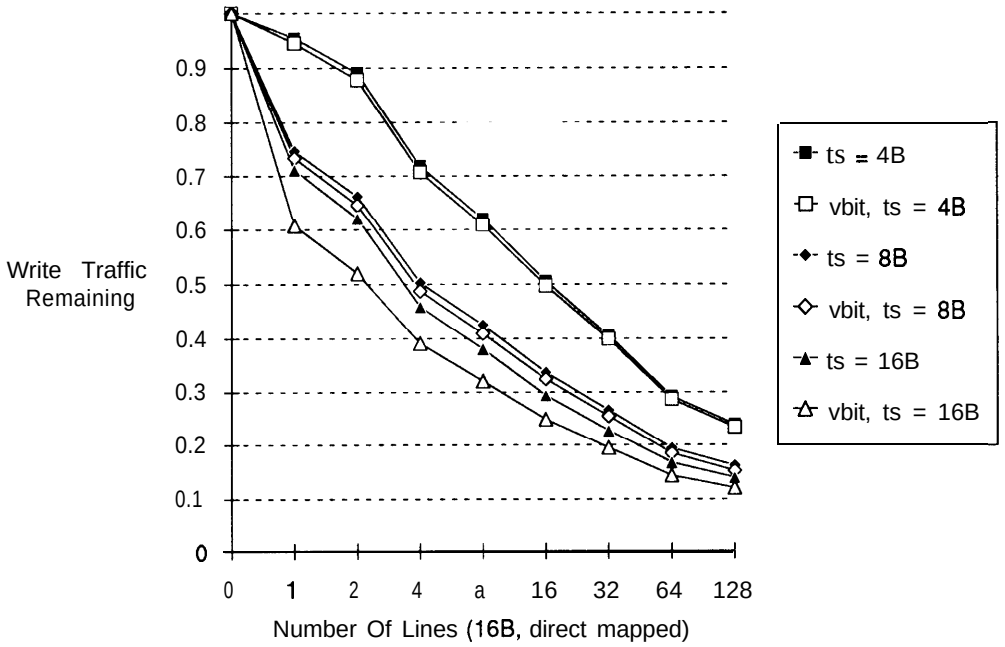

Figure 4: Effect Of Transfer Scheme On Write Traffic Reduction (Non-Sweep Benchmark Set)

numbers of lines is unfair. They must be compared in terms of area requirements. In [MQF91] the authors use the notion of register bit equivalent (rbe) as a unit of area to develop and validate area models for register files and caches. We use this model (see Appendix C) to look at organization and area trade-offs of ***write caches.*** Figure 7 compares various line sizes for a 4 byte transfer size for the non-sweep benchmark set while using area in rbes instead of number of lines. Now when comparing with an equal amount of area, the line size of 4 bytes outperforms the 16 byte lines. Figure 8 shows that an 8 byte transfer and line size performs initially better than the 16 byte transfer and line size. Looking at the line size tradeoffs, it does not seem profitable to have line sizes larger than the transfer size for the small ***write caches*** we are investigating.

2.4 Associativity

Some of the write traffic is due to mapping conflicts. A common technique to reduce mapping conflicts in caches is to use associativity. We attempt to reduce the conflicts by making ***the write cache*** four-way set associative with least recently used (LRU) replacement. LRU associativity is useful because it increases the probability that an object with high temporal locality remains in the cache, however, there must be objects which have high temporal locality. The sweep benchmark set has previously shown little temporal locality, so as expected, four-way LRU associativity proved to have little value (Figure 9).

The non-sweep benchmark set exhibits temporal locality, and as expected benefited from the LRU four-way set associativity (Figure 10). For few lines, the LRU four-way set associative ***write cache*** performed worse. This is because there were more highly temporal objects active at one time than the number of storage locations. A four-way set associative cache requires more in area than a comparable direct-mapped cache. So when area verses performance is plotted (Figure 11, 12, and 13), direct-mapped is initially better. The initial overhead and the anomolus behavior make the the LRU four-way set associative cache a poor choice for a very small cache. However, as area increases the LRU four-way set associative organization eventually outperforms direct-mapped for the non-sweep benchmark set.

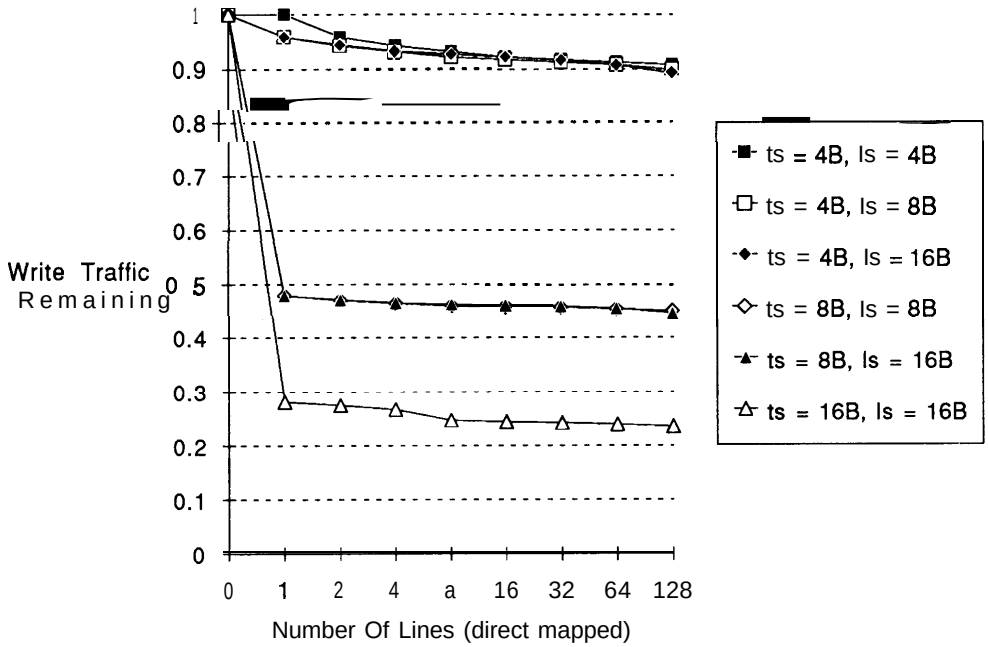

Figure 5: Effect Of Line Size On Write Traffic Reduction (Sweep Benchmark Set)

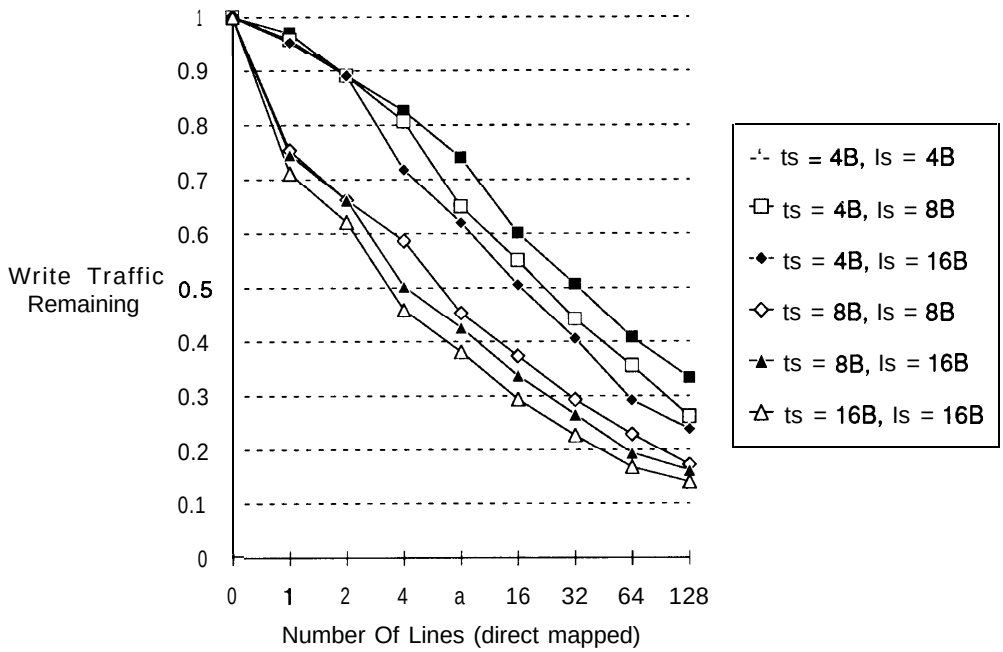

Figure 6: Effect Of Line Size On Write Traffic Reduction (Non-Sweep Benchmark Set)

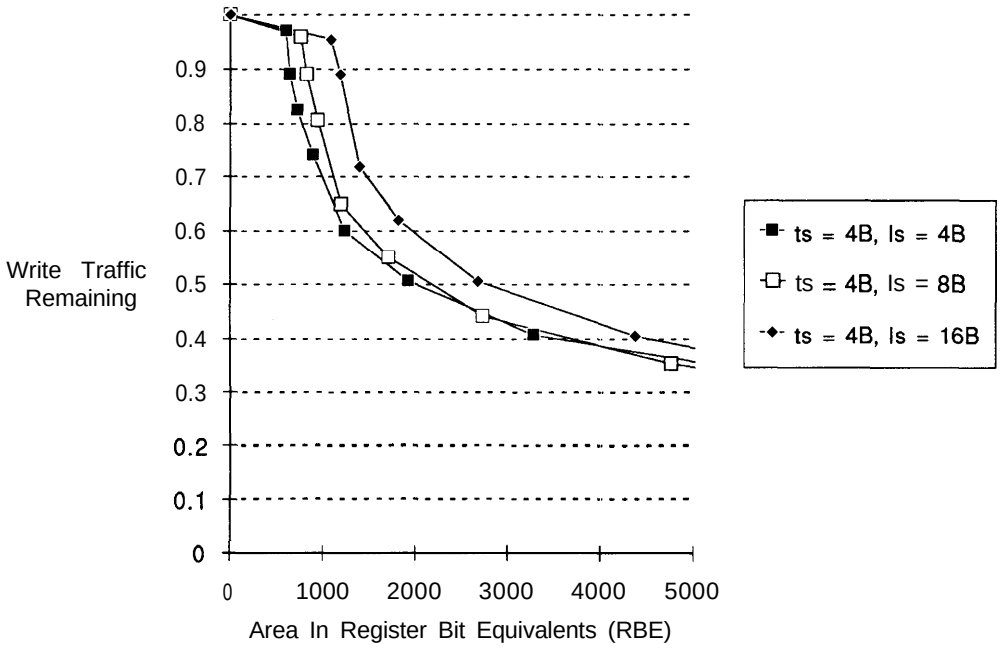

Figure 7: Area Effect Of Line Size On Write Traffic Reduction (Non-Sweep Benchmark Set)

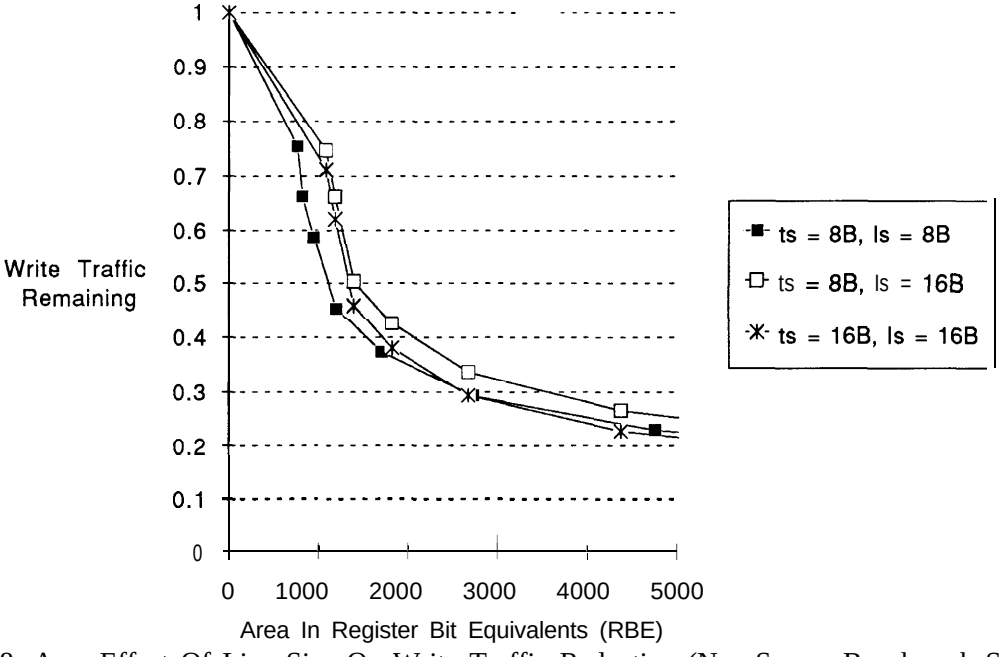

Figure 8: Area Effect Of Line Size On Write Traffic Reduction (Non-Sweep Benchmark Set)

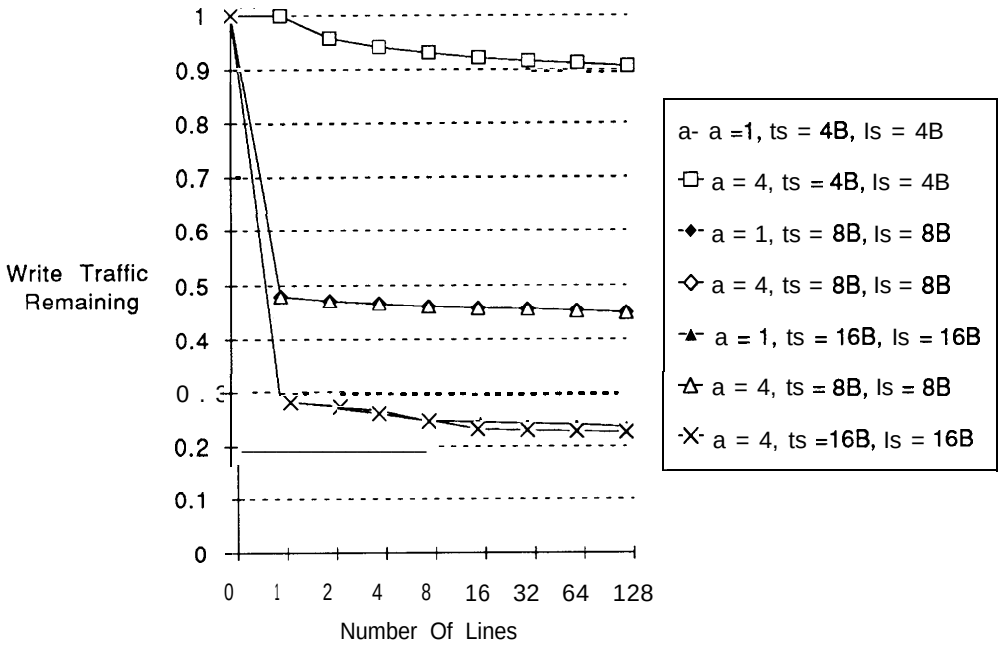

Figure 9: Effect Of Associativity On Write Traffic Reduction (Sweep Benchmark Set)

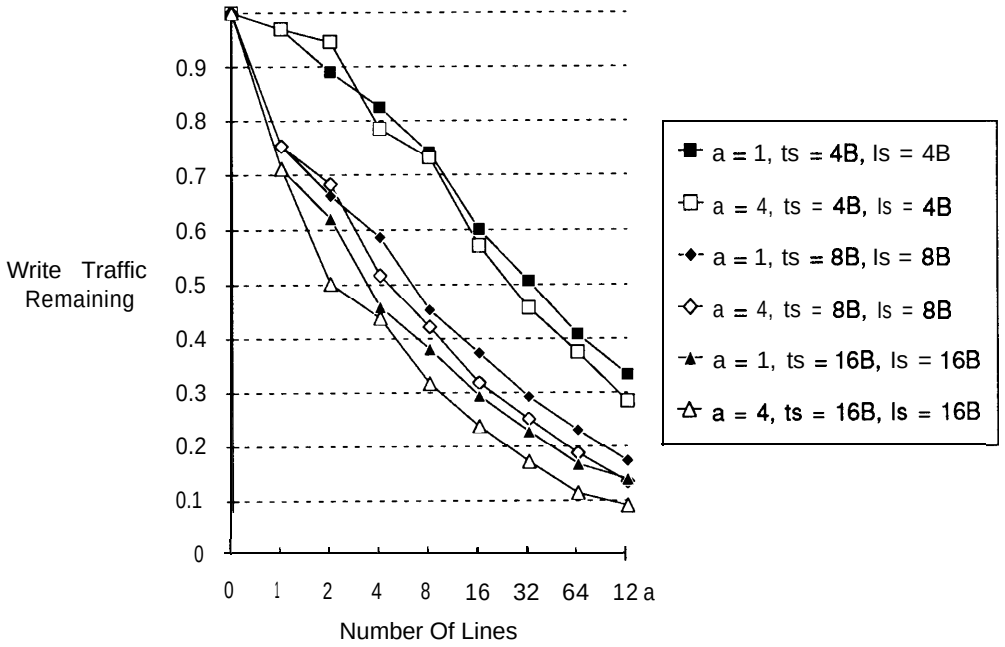

Figure 10: Effect Of Associativity On Write Traffic Reduction (Non-Sweep Benchmark Set)

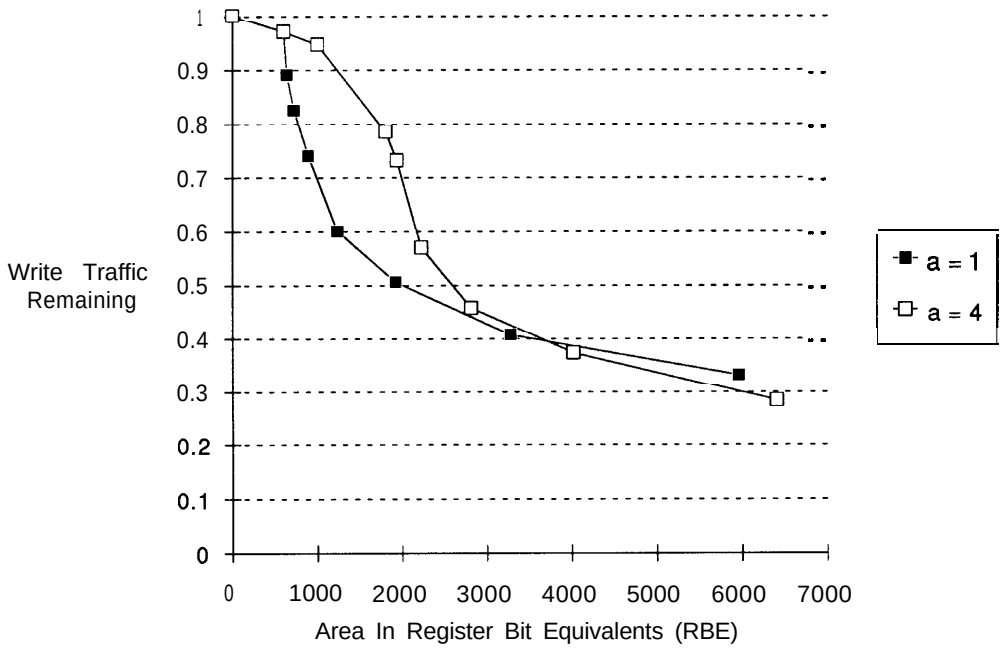

Figure 11: Area Effect Of Associativity On Write Traffic Reduction (ts = 4B, ls = 4B) (Non-Sweep Benchmark Set)

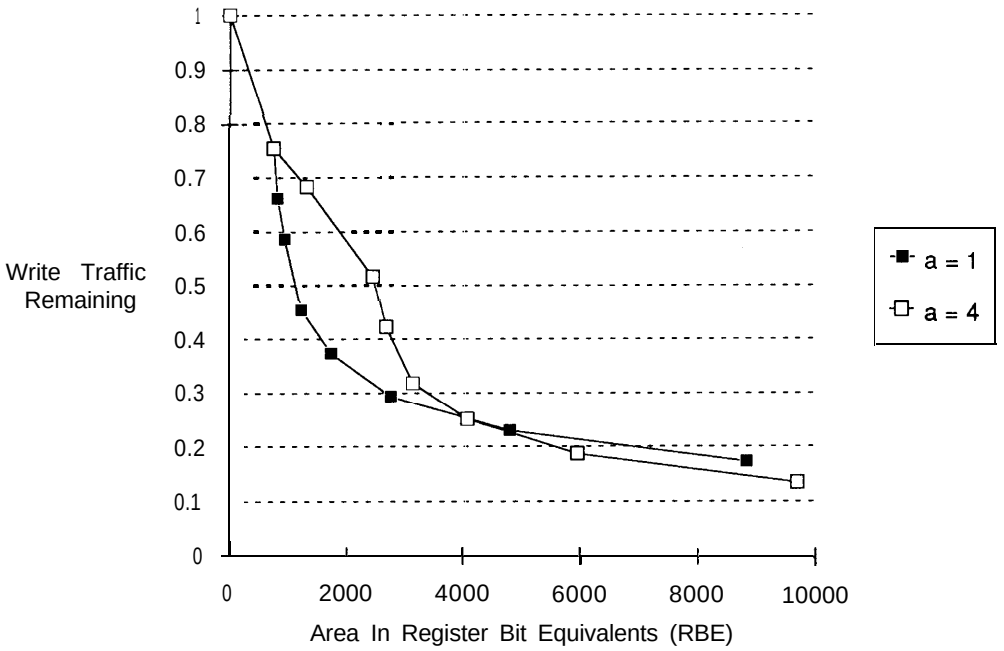

Figure 12: Area Effect Of Associativity On Write Traffic Reduction (ts = 8B, ls = 8B) (Non-Sweep Benchmark Set)

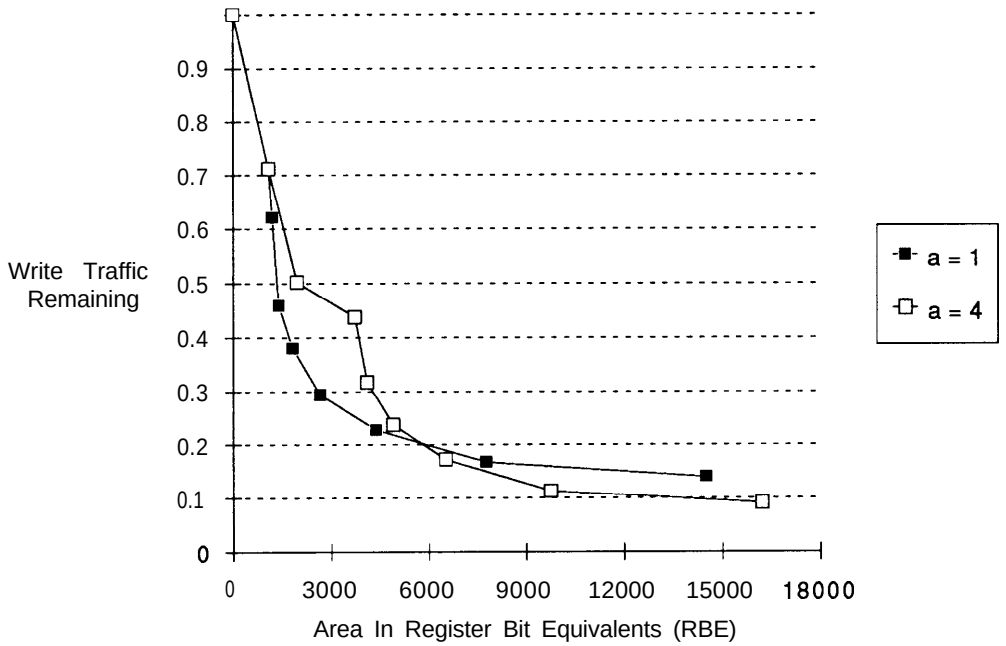

Figure 13: Area Effect Of Associativity On Write Traffic Reduction (ts = 16B, ls = 16B) (Non-Sweep Benchmark Set)

3 Consistency and Flushing

3.1 Flushing Interval

Previously we have been ignoring the effects of process switching; only at the end of the simulation of an application would ***the write cache*** be flushed. Flushing can increase the write traffic because items that would be written to again are no longer present, thus decreasing the effect of temporal locality. The benchmark sets were simulated with a flushing interval of 1000 and 5000 writes. If flushing occurs every 1000 writes and if 10% of the instructions are writes, that corresponds to a process switch interval of 10,000 instructions. Ten thousand instructions is usually a small time slice, so any degradation due to normal process switch flushing should be less. The flushing caused no performance loss for the sweep benchmark set, since it had very little temporal locality. The flushing became noticable for the non-sweep benchmark set for sizes of 32 lines and above (Figure 14). However, the degradation was small, and there was practically no degradation when the flush interval was more reasonably set to 5000 writes.

3.2 Enforcing Data Consistency

Like a write buffer, a ***write cache*** has to deal with data consistency conflicts that might occur between reads and buffered writes. Part of the valid data for a read request may be in the write buffer or *write* ***cache,*** so there must be some mechanism to insure that the correct results are returned to the read requestor. Figure 15 and 16 show the probabilty that a data cache read miss is in ***the write cache. The*** data cache is direct-mapped, write-through, no allocate on write miss, with 16 byte lines. The data cache size of OK bytes represents the case when the ***write cache*** is between the processor and the data cache. (This is useful when trying to reduce the write traffic to the first-level cache.)

A very simple scheme to insure data consistency would be to allow the reads to bypass the *write cache.* If there were a conflict, the returning read results would be nullified, and the ***write cache*** would flush

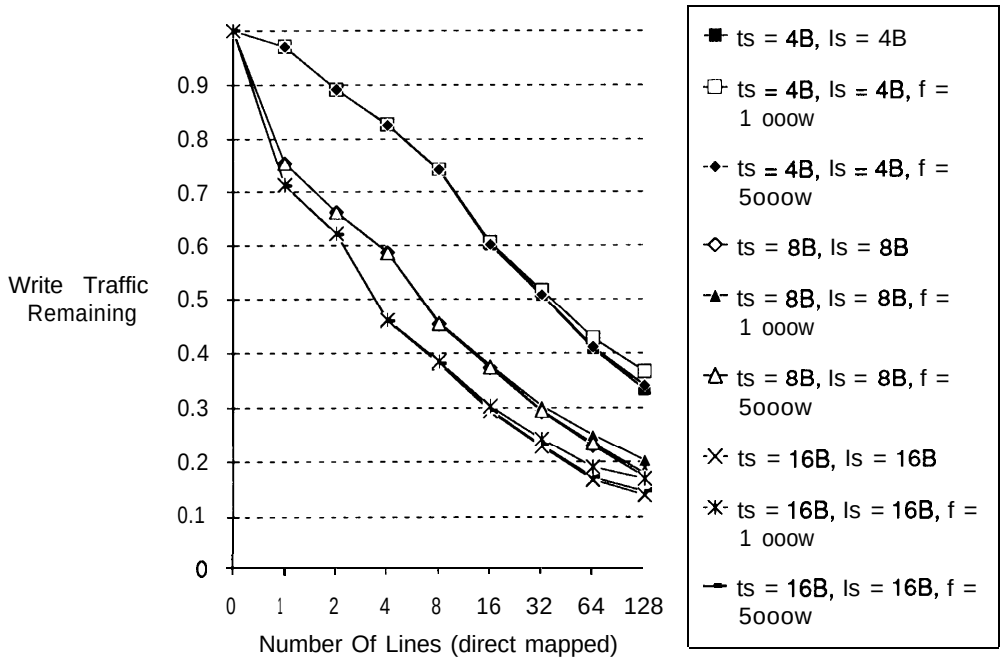

Figure 14: Effect Of Flushing On Write Traffic Reduction (Non-Sweep Benchmark Set)

the the conflicting entries to the next level, and then the read request would be retried. However, as the number of reads which are partially or entirely in the ***write cache*** become significant, the mechanism to transparently insure consistency must must be made more complex, so handling of conflicts causes little additional delay to prevent significant increase in the average read miss penalty. A slightly more complex scheme would have the read results returned to the requestor, and then the ***write cache*** would copy any conflicting entries to the read requestor. A more complex scheme might be one which merges (with no additional delay) the requested results and any relavent data in ***the write cache,*** as the data is being returned to the requestor.

The write allocation policy of the first-level data cache can have a significant effect on the number of consistency conflicts. Figure 17 shows that changing the write policy in write-though caches from no allocate on write miss to allocate on write miss will significantly reduce the consistency conflicts. Consistency conflicts occur more often than expected because locations usually are written to (initialized) before the first time they are ever read. So if the data cache doesn't use a write allocate on write miss policy, the chance for conflicts is higher.

Like write buffers, ***write caches*** have another data consistency problem; I/O register writes can have side effects at other I/O register addresses. Therefore I/O register writes must complete in the order of issue and before any I/O reads can be issued. Therefore, operating system device drivers must explicitly wait for a I/O register write to complete before reading device registers. On the DECstation5000 this is accomplished by calling a *wbFlush* routine to insure the write buffer is free of any I/O writes [Eng90].

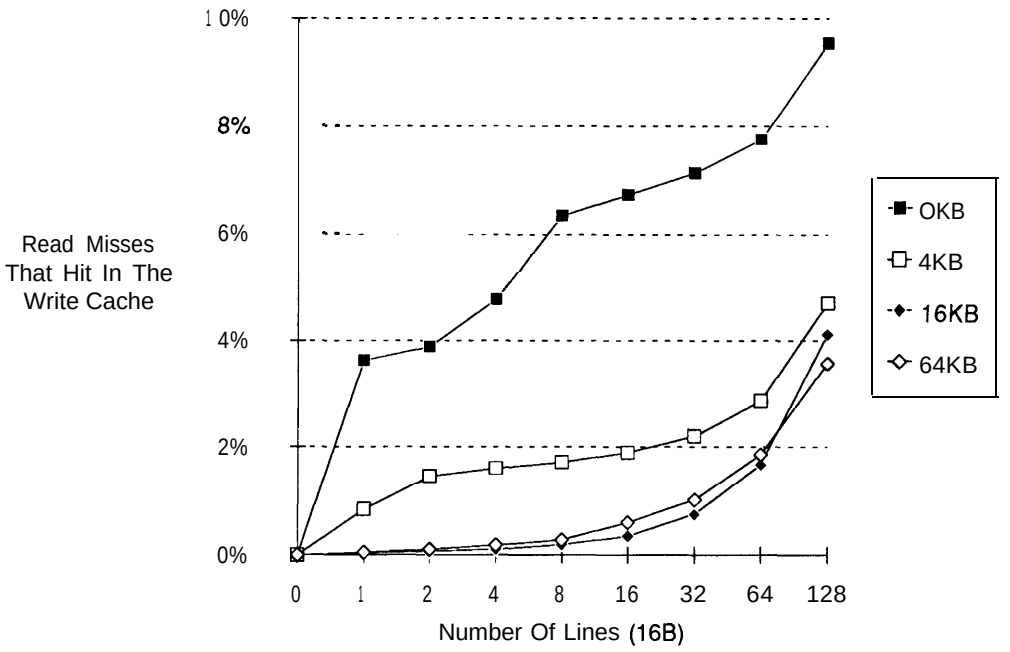

Figure 15: Effect Of First-Level Cache Size On Hitting In The Write Cache (Sweep Benchmark Set)

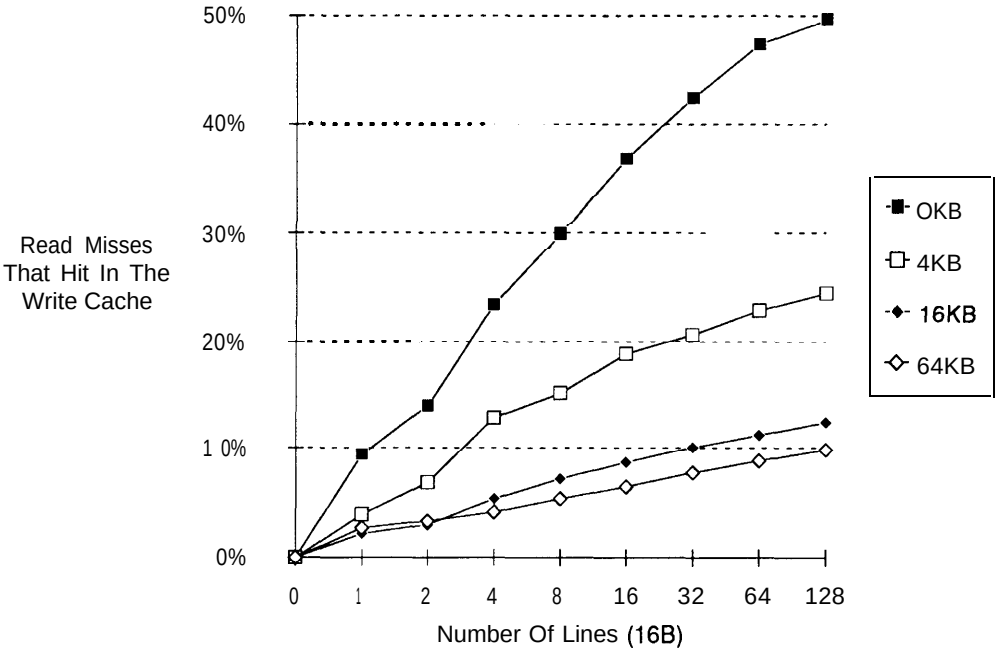

Figure 16: Effect Of First-Level Cache Size On Hitting In The Write Cache (Non-Sweep Benchmark Set)

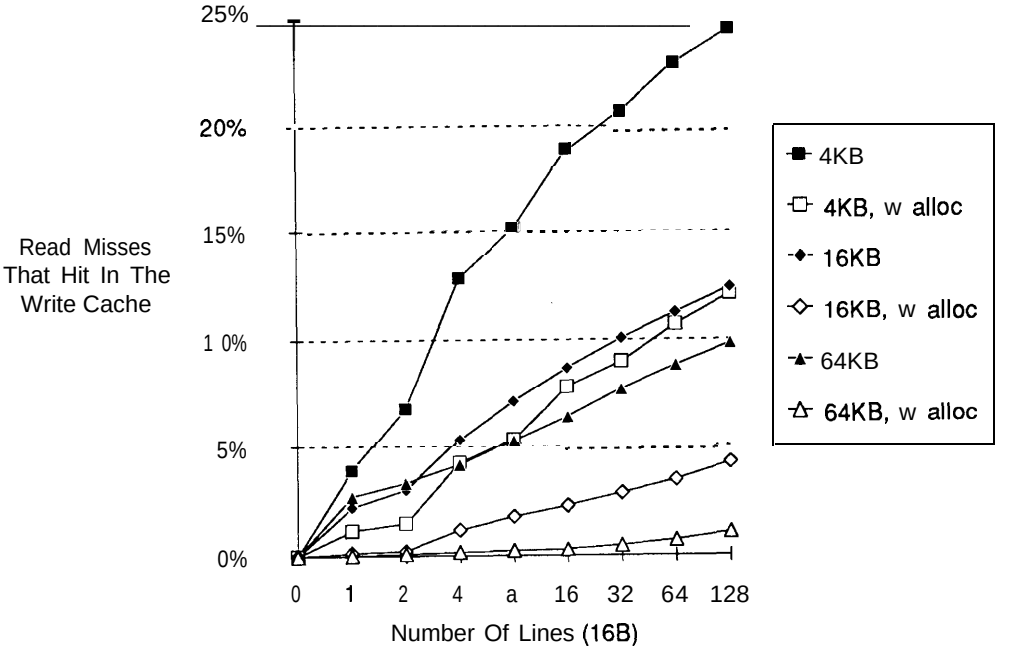

Figure 17: Effect Of First-Level Cache Write Allocation Policy On Hitting In The Write Cache (Non-Sweep Benchmark Set)

4 Conclusion

Small ***write caches*** can significantly reduce the write traffic to the first write-back level after the processor's register set. Systems that would benefit from reduced write traffic to the first write-back level would benefit from using a ***write cache*** instead of a write buffer. The temporal and spatial locality of writes is very important in determining what organization ***the write cache*** should have. The organizational observations are in the following Table 1.

	Applications whose writes have low temporal locality and high spatial locality	Applications whose writes have high temporal locality	Suggested General Solution
Number Of Lines	1 line is sufficient	most important, for approximately a 50% write traffic reduction (16-32 lines with ts=4B, 4-8 lines with ts=8B,16B)	minimum of 4 lines
Transfer Size	most important, if not $> 4B$ write cache will be ineffective, $16B$ works well	important if don't have enough lines, but $> 8B$ has little benefit unless use vbit transfer scheme	$8B$
Transfer Scheme	simple	simple, unless have transfer size $> 8B$ and only a few lines	simple
Line Size	= Transfer Size	= Transfer Size	= Transfer Size
Associativity	direct-mapped	direct-mapped, unless size becomes large enough that conflict misses dominate capacity misses	direct-mapped

Table 1: Summary Of Organizational Parameters For Write Caches

References

[Eng90] Workstation System Engineering. *DECstation 50001200 KN02 System Module Functional Specification Revision 1.3*. Digital Equipment Corporation, 1990.

[FKH87] John Fu, J. Keller, and K. Haduch. Aspects of *the* VAX *8800 C* Box Design. ***Digital Technical Journal,*** (4):41–51, February 1987.

[Jou90] Norman Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. In ***Conference Proceedings, The 17th Annual Symposium on Computer Architecture,*** pages 364-373, May 1990.

[MQF91] Johannes Mulder, N. Quach, and M. Flynn. An Area Model for On-Chip Memories and its Application. ***IEEE Journal*** *of* ***Solid-State Circuits,*** 26(2):98–106, February 1991.

[Smi82] A. J. Smith. Cache Memories. ***ACM Computing*** Surveys, 14(3):473–530, September 1982.

A Write Traffic Penalties

The rate at which writes can be serviced in the memory hierarchy can become the upper bound on performance. Assuming that writes take one access (or can be pipelined), Figure 18 shows the first write-back level's access time at which performance is limited by the rate of servicing write requests (100% occupancy due to writes). As the writes per instruction or the issue rate increases the write service request rate increases, and the access time of the first write-back level of cache (or main memory) must decrease.

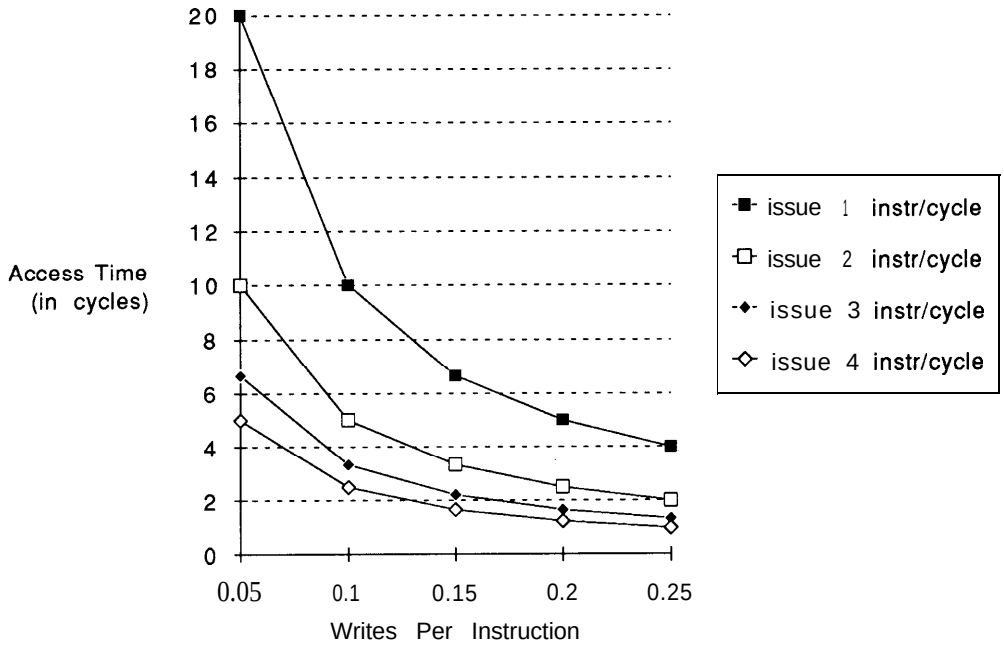

Figure 18: Access Time of the Write-Back Level of Cache (or Main Memory) Which Would Cause 100% occupancy

The write traffic can not only limit execution rate by causing the processor to stall because the write buffers are full, but also by requiring read requests to wait for pending writes. As a result, the read miss penalty (= ***access time + transfer time + busy time***) will increase, further limiting performance. The increase in the read miss penalty is shown in Figure 19. This is assuming that a read request can bypass all pending writes except for the one in progress, and that if memory is busy with a write, the write is halfway through ***an*** access. ***Thus the*** busy time due to writes equals ***occupancy*** x $\frac{accesstime}{2}$.

Note that by reducing the write traffic, the occupancy due to writes will reduce. By reducing the occupancy, the read miss penalty is also reduced.

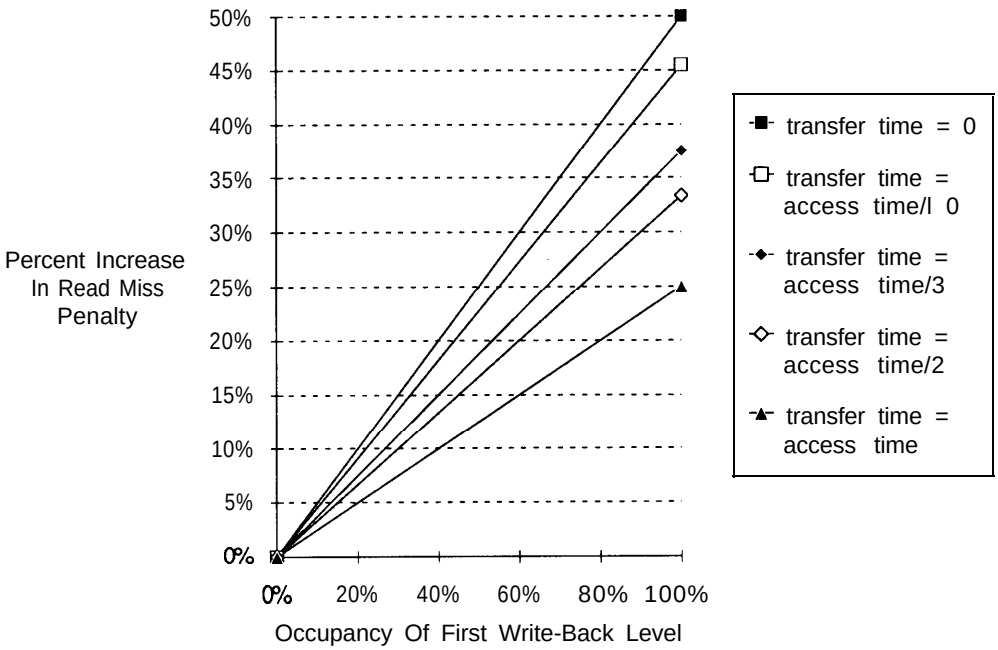

Figure 19: Increase In Read Miss Penalty Due To Servicing Writes

B Benchmarks

Trace driven simulation produced the presented results. The architecture simulated was essentially the MIPS R2000/R3000. Note that the R2000/R3000 does not have 64-bit floating point loads and stores, a 64-bit load or store is made by using two 32-bit transfers. The following C benchmarks were optimized with the Ultrix 3.1 C compiler with optimization level 02. The code executed in the application code, string routines, and printf routines was simulated, while the code executed in system calls, scanf routines, and math libraries was not.

Name	Description	Instr. (10^6)	% loads	% stores
`fft`	fast fourier transform - 1024x1024 2-D fft	10.1	41.6	17.1
`lloops`	composite of the 14 Livermore Loops	0.09 16	42.9	17.3

Table 2: Benchmarks Whose Writes Were Dominated By Sweeping Through Large Structures (Sweep Benchmark Set)

Name	Description	Instr. (10^6)	% loads	% stores
`gas`	gnu assembler - assembling a 1800 line file	6.54	23.0	13.3
`gcc1`	gnu C compiler version 1.36 - compiling (and optimizing) to assembly code a 1500 line C program	43.6	23.4	13.1
`gdiff`	gnu diff - comparing two 1112 line files	1.13	23.7	13.0
`grep`	grep - searching a file for regular expression	0.191	18.5	13.3
`nettuner`	nettuner - reading a netlist of a 4x4 multiplier and padding the circuit delays	12.6	22.5	15.0
`spice3`	circuit simulator - simulating a Schottky TTL edge-triggered register	186	43.7	13.7
`TeX`	document preparation system - formatting of a 14 page technical report	82.0	22.6	15.9

Table 3: Benchmarks Whose Writes Were Not Dominated By Sweeping Through Large Structures (Non-Sweep Benchmark Set)

C Area Model

In [MQF91], the authors developed and validated an area model for caches and register files. They also used the area model to look at cache organization trade-offs as a function of area. Instead of making the area unit technology dependent (by making the area unit a measure of square micrometers), the authors had made the area unit technology independent. The area unit of measure is a register bit equivalent **(rbe).** The rbe area unit equals the area of a register cell. The area models take into account the area needed for data bits, tag bits, status bits, overhead logic (drivers and comparitors), and consider the effects of bandwidth requirements of memory cells.

Area for a set associative cache is:

$$area = control + data + tags + status$$

$$= 195 + 0.6 \times overhead_1 \times size_b + 0.6 \times overhead2 \times tsbits_b \text{ rbe}$$

Where:

$size_b$ = data bits in the cache

$tsbits_b$ = tag and Status bits in the cache = ts_b x $lines = ts_b \times tags$

ts_b = status bits per line + tag bits per line

$$= (1 + dv \times tunits + log_2(\frac{2^{30} words \times assoc}{\frac{size_b}{32\frac{b}{word}}}))$$

Note: We are assuming the processor addresses 2^{30} words. Also there was a mistake in the ts_b equation published in [MQF91], the 32 $\frac{b}{word}$ had been omitted.

dv = dirty and valid bits per transfer unit

$$tunits = \frac{linesize_b}{subblocksize_b}$$

$$tags = \frac{size_b}{linesize_b}$$

$$overhead_1 = 1 + \frac{6 \times assoc}{tags} + \frac{6}{linesize_b \times assoc}$$

$$overhead2 = 1 + \frac{12 \times assoc}{tags} + \frac{6}{ts_b \times assoc}$$

The area model is slightly modified for *write caches*.

$tunits = \frac{linesize_b}{8b}$ since the subblock size is a byte.

$$ts_b = (tunits + log_2(\frac{2^{30} words \times assoc}{\frac{size_b}{32\frac{b}{word}}}))$$

The authors of the model also say there is a single valid bit per line, in addition to the valid bits per subblock. We think that is unneccesary, since there is a valid bit per transfer unit. The dv is 1, since there is a single combined dirty/valid bit per subblock; in a *write cache,* reads are never allocated thus the valid bit also acts as the dirty bit.

The area requirements for various *write cache* sizes and organizations is shown in Figure 20. For a frame

of reference, a 32x32b, 2 read port, 1 write port register file occupies 3200 rbe. It is interesting to note that as the number of lines increase, the overhead of associativity has less effect on total cache size. However, for small caches, the overhead of associativity is significant.

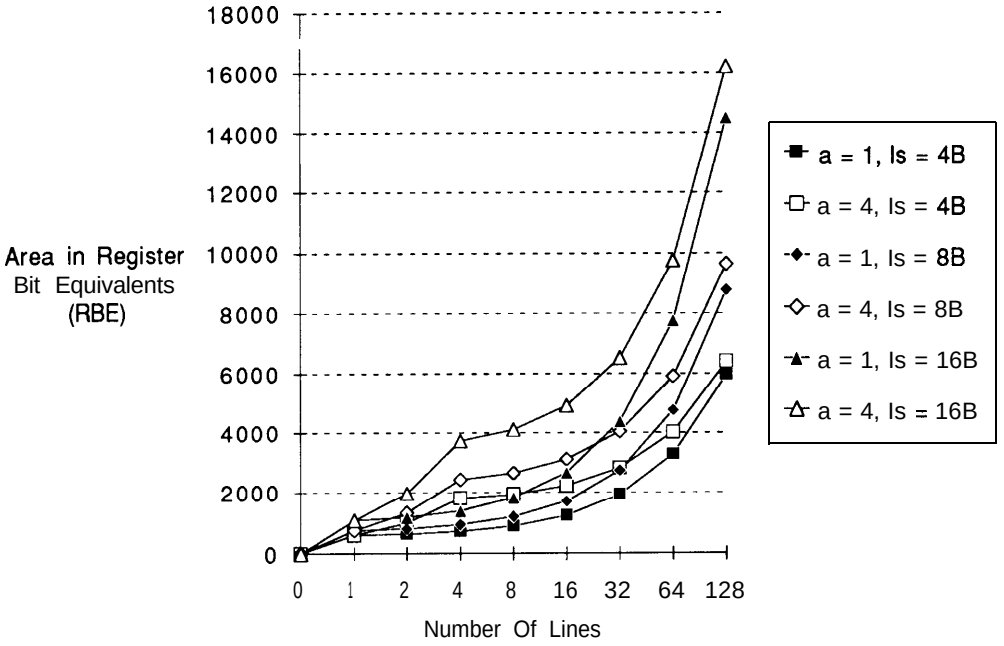

Figure 20: Area Requirements For Various ***Write Cache*** Sizes And Organizations