DECEMBER 1991

WRL Research Report 91/12

Cache Write Policies and Performance

Norman P. Jouppi

digital **Western Research Laboratory** 250 University 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, WRL-2
250 University 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`
Internet:	`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.

Cache Write Policies and Performance

Norman P. Jouppi

December, 1991

Abstract

This paper investigates issues involving writes and caches. First, tradeoffs between write-through and write-back caching when writes hit in a cache are considered. A mixture of these two alternatives, called *write caching* is proposed. *Write caching* places a small fully-associative cache behind a write-through cache. A write cache can eliminate almost as much write traffic as a write-back cache. Second, tradeoffs on writes that miss in the cache are investigated. In particular, whether the missed cache block is fetched on a write miss, whether the missed cache block is allocated in the cache, and whether the cache line accessed is invalidated are considered. Depending on the combination of these polices chosen, the entire cache miss rate can vary by a factor of two on some applications. Furthermore, the combination of no-fetch-on-write and write-allocate can provide better performance than cache line allocation instructions. Finally, the traffic at the back side of write-through and write-back caches with various parameters is characterized.

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Table of Contents

1. Introduction **1**
2. Experimental Environment **2**
3. Write Hits: Write-Through vs. Write-Back **2**
3.1. Increasing Write-Back Cache Bandwidth **8**
3.2. Reducing Write-Through Cache Traffic **9**
3.3. Summary: When to Choose Write-Back or Write-Through **13**
4. Write Misses: Fetch-on-Write vs. Write-Validate vs. Write-Around vs. Write-Invalidate **14**
5. Traffic Out the Back of the Cache **24**
5.1. Traffic Measured in Transactions **24**
5.2. Traffic Measured in Bytes **25**
6. Conclusions **31**
Acknowledgements **32**
References **32**

List of Figures

Figure 1: Write-back vs. write-through cache behavior for 8KB caches 4
Figure 2: Write-back vs. write-through cache behavior for 16B lines 4
Figure 3: Direct-mapped write-through and write-back pipelines 7
Figure 4: Delayed write method for write-back caches 8
Figure 5: Coalescing write buffer merges vs. CPI 9
Figure 6: Write cache organization 10
Figure 7: Write cache absolute traffic reduction 11
Figure 8: Write cache traffic reduction relative to a 4KB write-back cache 12
Figure 9: Relative traffic reduction of a write cache vs. write-back cache size 12
Figure 10: Write misses as a percent of all misses vs. cache size for 16B lines 14
Figure 11: Write misses as a percent of all misses vs. line size for 8KB caches 15
Figure 12: Write miss alternatives 15
Figure 13: Write miss rate reductions of three write strategies for 16B lines 18
Figure 14: Total miss rate reductions of three write strategies for 16B lines 20
Figure 15: Write miss rate reductions of three write strategies for 8KB caches 21
Figure 16: Total miss rate reduction of three write strategies for 8KB caches 22
Figure 17: Relative order of fetch traffic for write miss alternatives 23
Figure 18: Components of traffic vs. cache size 25
Figure 19: Components of traffic vs. cache line size 26
Figure 20: Percent of victims with dirty bytes vs. cache size for 16B lines 27
Figure 21: Percent of bytes dirty in a dirty victim vs. cache size for 16B lines 27
Figure 22: Percent of bytes dirty per victim vs. cache size for 16B lines 28
Figure 23: Percent of victims with dirty bytes vs. line size for 8KB caches 29
Figure 24: Percent of bytes dirty in a dirty victim vs. line size for 8KB caches 29
Figure 25: Percent of bytes dirty per victim vs. line size for 8KB caches 30

List of Tables

Table 1: Test program characteristics **2**
Table 2: Advantages and disadvantages of write-through and write-back caches **7**
Table 3: Hardware requirements for high performance write-back and write-through caches **13**

1. Introduction

Most of the extensive literature on caches has concentrated on read issues (e.g., miss rates when treating stores as reads), or writes in the context of multiprocessor cache consistency. However, uniprocessor[1] write issues are in many ways more complicated than read issues, since writes require additional work beyond that for a cache hit (e.g., writing the data back to the memory system).

The cache write policies investigated in this paper fall into two broad categories: write hit policies, and write miss policies.

Unlike instruction fetches and data loads, where reducing latency is the prime goal, the primary goal for writes that hit in the cache is reducing the bandwidth requirements (i.e., write traffic). This is especially important if the cycle time of the CPU is faster than that of the interface to the second-level cache, and if multiple instruction issue allows store traffic approaching one per cycle to be sustained in many applications. The write traffic into the second-level cache primarily depends on whether the first-level cache is *write-through* (also called *store-through*) or *write-back* (also called *store-in* or *copy-back*). Write-back caches take advantage of the temporal and spatial locality of writes (and reads) to reduce the write traffic leaving the cache.

Write miss policies, although they do affect bandwidth, focus foremost on latency. Write miss policies include three semi-dependent variables. First, writes that miss in the cache may or may not have a line allocated in the cache (*write-allocate* vs. *no-write-allocate*). If a cache uses a no-write-allocate policy, when reads occur to recently written data, they must wait for the data to be fetched back from a lower level in the memory hierarchy. Second, writes that miss in the cache may or may not fetch the block being written (*fetch-on-write* vs. *no-fetch-on-write*). A cache that uses a fetch-on-write policy must wait for a missed cache line to be fetched from a lower level of the memory hierarchy, while a cache using no-fetch-on-write can proceed immediately. Third, writes that miss in the cache may simply invalidate the cache line accessed and pass the data written on to lower levels in the memory hierarchy (*write-invalidate* vs. *no-write-invalidate*). Different combinations of these three variables can result in a 2:1 range in cache miss rates for some applications.

Out of the hundreds of papers on caches in the last 15 years [15, 16], Smith [13] was the only paper to exclusively deal with write issues. This paper discussed write buffer performance for write-through caches, but did not investigate merging of pending writes to the same cache line by a write buffer. Smith [14] and Goodman [7] both have a section on write-back versus write-through caching, but they study mixed first-level caches with traces under a million references. Among the more recent work in uniprocessor cache issues, Agarwal [1] and Hill [8] assumed write references were identical to read references in their analysis. Przybylski [11] includes write overheads in his analysis, but only considers the case of write-back caches at all levels. Write miss policies have been even less investigated. Almost all of the known results in the literature have been for the combination of write-allocate and fetch-on-write. The VAX 11/780 [2] and 8800 [3] were notable exceptions to this and used no-write-allocate. No known results in the literature compare the performance of different write miss policies.

---

[1]By uniprocessor we include non-coherency issues in multiprocessor cache memories, as well as uniprocessor cache memories.

This paper investigates write policies in the context of a modern memory hierarchy. Two or more levels of caching are assumed, although the data in the paper is for the effects of these policies on the first-level cache performance. Separate instruction and data caches are assumed at the first level, since these are necessary for superscalar and other types of high performance machine design.

Section 2 briefly describes the simulation environment and benchmarks used in this study. Section 3 investigates write hit tradeoffs between write-back and write-through caching, as well as ways of reducing write-through traffic. Policies for write misses, specifically fetch-on-write, write-allocate, and write-invalidate are investigated in Section 4. The traffic components at the back side of write-through and write-back caches are studied in Section 5. Section 6 summarizes the results of the paper.

2. Experimental Environment

The results in this paper were obtained by modifying a simulator for the MultiTitan [9] architecture. The MultiTitan architecture does not support byte loads and stores, so byte writes appear as word read-modify-writes. However, the number of byte operations in the programs studied are insignificant, so this does not significantly affect the results presented. Each experiment involved simulating the benchmarks, and not analyzing trace tapes.

The characteristics of the test programs used in this study are given in Table 1. Although six is a small number of benchmarks, the programs chosen are quite diverse, with two numeric programs, two CAD tools, and two Unix utilities. However, operating system execution, transaction-processing code, commercial workloads (e.g., COBOL), and multiprocessing were beyond the scope of this study. The benchmarks used are reasonably long in comparison with most traces in use today.

program name	dynamic instr.	data reads	data writes	total refs.	program type
ccom	31.5M	8.3M	5.7M	45.5M	C compiler
grr	134.2M	42.1M	17.1M	193.4M	PC board CAD tool
yacc	51.0M	12.9M	3.8M	67.7M	Unix utility
met	99.4M	36.4M	13.8M	149.7M	PC board CAD tool
linpack	144.8M	28.1M	12.1M	185.5M	numeric, 100x100
liver	23.6M	5.0M	2.3M	31.0M	Livermore loops 1-14
total	484.5M	132.8M	54.8M	672.8M	

Table 1: Test program characteristics

3. Write Hits: Write-Through vs. Write-Back

When a write hits in a cache, two possible policy choices exist. First, the data can be written both into the cache and passed on to the next lower level in the memory hierarchy. This policy is called write-through. A second possible policy on write hits is to only write the data to the first-level cache. Only when a dirty line (i.e., a line that has been written to) is replaced in the cache is the data transferred to a lower level in the memory hierarchy. This policy is called write-back. Write-back caching takes advantage of the locality of reference of writes to reduce the amount of write traffic going to the next lower level in the memory hierarchy.

A first dimension of comparison between write-through and write-back caching is the write traffic out of the cache. Figure 1 shows the percentage of writes to already dirty cache lines for 8KB caches of varying line sizes. Note that for the case of write hits (i.e., ignoring read and write miss traffic):

$$\textit{write back transactions} = \textit{\# of writes} - \textit{\# of writes to already dirty lines}$$

and

$$\textit{fraction of writes to already dirty lines} = 1 - \frac{\textit{write back transactions}}{\textit{write through transactions}}$$

The percentage of writes to already dirty cache lines was used instead of measuring the write-back traffic for a number of reasons. Foremost among these is the *cold stop* problem. For caches which are larger than a program's working set, at the end of program execution the majority of cache lines written by the program can still be resident in the cache. This makes collection of information on write-back traffic more difficult. A second reason is that the performance of some cache organizations depends on the percentage of writes which are to already dirty cache lines, as we will see in Section 3.1. Finally, note that the number of writes to already dirty lines is not the same as the reduction in write traffic in bytes, nor the utilization of the write bus, which may be pipelined or have some piece-wise linear latency in terms of write size. These issues are covered in Section 5.

Figure 1 shows that as the line size increases, the odds that more than one write will hit the same cache line increase. If dirty write-back cache lines are written back in their entirety this gives the percent reduction in write traffic obtained by write-back caching. If there are dirty bits in the write-back cache on a smaller granularity than the entire cache line, and the write-back port width is less than a cache line wide, the width of the dirty bit granularity and write-back port width should be used in comparisons.

Linpack and *liver* have the worst write-back cache effectiveness for short lines in Figure 1. This is because the 8KB cache is too small to contain their working set, so lines that are written get replaced in the cache before being written again. Note that their behavior for 4B and 8B lines are nearly identical. This is because these benchmarks use double-precision (8B) data, so in both cases each line only gets one write before being replaced. With each doubling in line size beyond 8B, the number of writes remaining for *linpack* and *liver* approximately halves, since the number of double-precision values per cache line is doubling. On average over the six benchmarks, the write-back cache is able to remove the majority of writes, even for small line sizes.

Figure 2 shows the percentage of writes to already dirty cache lines in various sizes of write-back caches with 16B lines. Assuming an entire line is written back if any part of it is dirty, this also gives the percentage of write traffic removed by write-back caching. *grr*, *yacc*, and *met* experience 80% or greater reductions in write traffic by the use of a write-back cache. Therefore these benchmarks have very good write locality of reference. On the other hand, *linpack* and *liver* operate in a sequential fashion through large matrices, and lines that are written are replaced in the cache before they can be used or written again except for cache sizes greater than 64KB. Part of the poor effectiveness of write-back caches on numerical codes is the bias of these codes towards vector machines without caches. For good operation on vector machines the longest streams possible were fetched from main memory, without dependencies or reuse of

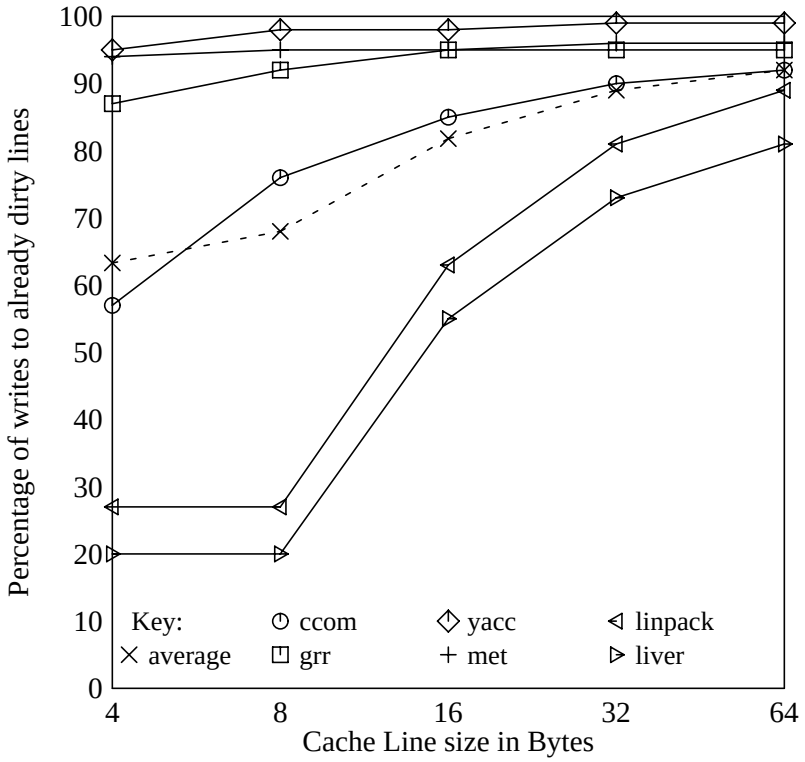

Figure 1: Write-back vs. write-through cache behavior for 8KB caches

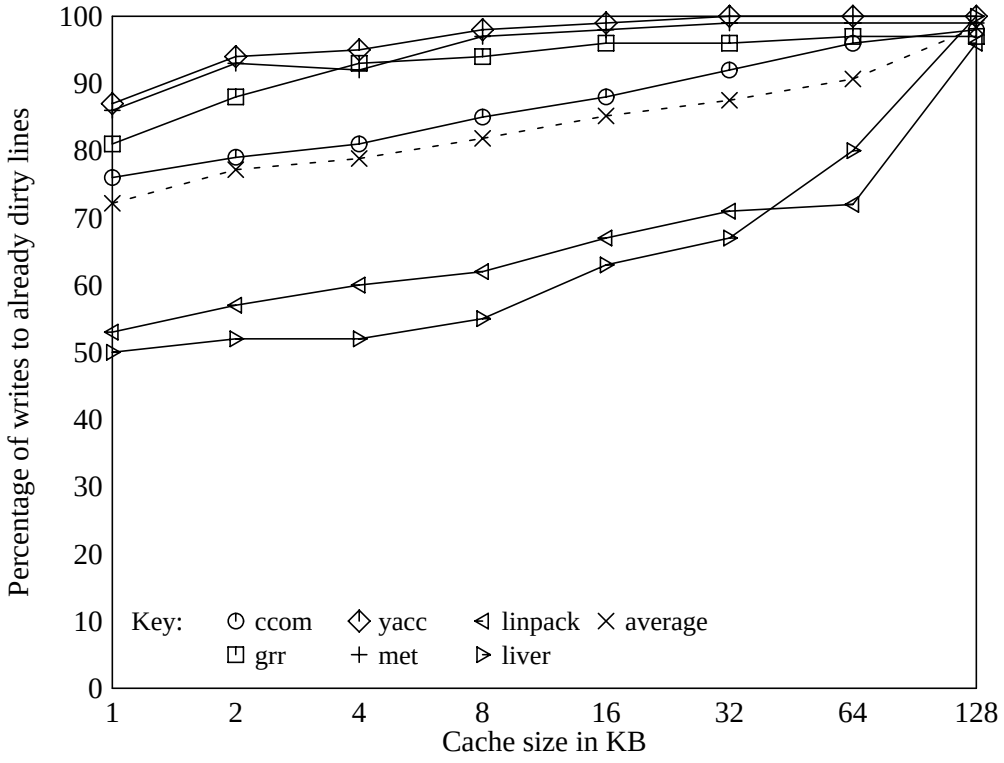

Figure 2: Write-back vs. write-through cache behavior for 16B lines

results. In general, as numeric and other programs are restructured to make better use of caches and vector register files, the usefulness of write-back caches will increase. For example, with block-mode numerical algorithms the percentage of write traffic saved should be significantly higher. For non-numeric programs, write-back caches can have significantly lower write traffic requirements than write-through caches. Figure 2 shows that write traffic decreases with increasing cache size, although even for 32KB caches *linpack* and *liver* still write a double-precision value less than two times on average while it is mapped in the cache.

A second dimension of comparison between write-back and write-through caching is the requirement for extra buffers. Both the write-through and write-back caches require at least a one-word buffer for adequate performance, and the write-through cache is in fact likely to require several buffer entries. A write-back cache requires a buffer entry to hold a dirty victim[2]. In the event of a miss a dirty victim can be transferred into the dirty victim buffer at the same time as the fetch of the requested word is begun. Then once the next lower level is ready to service another request, the dirty victim can be emptied out. Only in the case where the next lower level in the hierarchy is not pipelined and multiple misses with dirty victims occur in series would a dirty victim buffer with more than one entry be useful for a write-back cache. A write buffer for a write-through cache typically requires two to four entries [13]. Although the difference in implementation cost between a single-entry buffer and a two-entry or four-entry buffer was significant when buffers were implemented from MSI latch parts on printed-circuit boards, in a VLSI environment the wires to and from the buffer and the fixed overhead logic per buffer make the area difference between a four-entry write-buffer and a single-entry dirty victim buffer considerably less than 4 to 1. An excellent study of write buffer performance appeared in [13].

A third dimension of comparison is the ability of the cache to handle bursty write traffic entering the cache. If writes occur in large clusters, the write buffer of a write-through cache will fill up and the CPU will be forced to stall, with further writes progressing only at the rate handled by the next lower level cache. A write-back cache would be able to sustain a higher rate of writes, assuming the writes did not all miss in the cache and the victims were not all dirty. Burstiness can be an even more significant problem for machines that use register windows. When the window stack overflows, some of the register window frames must be dumped to memory. This can result in a series of 30 or more sequential stores. Also if the machine has CISC procedure call instructions which save a large number of registers on procedure calls, many sequential stores could occur. Similarly, if procedural register allocation methods which allocate registers on a per-procedure basis are used, large bursts of stores will result on procedure calls so that these registers can be saved. Our compilers [17] use global register allocation. This requires virtually no save and restore traffic on procedure calls, and so does not contribute to the burstiness of write traffic. In contrast to register saves where a long series of stores are back-to-back, the worst-case store traffic from most algorithms is that presented by block copy operations where stores and loads are interleaved. For very large block copies (i.e., sustained bursty writes) write-back and write-through caches have similar performance. This is because they are both limited by the write bandwidth of the memory system.

[2]A *victim* is the cache entry which is replaced on a miss.

The fourth dimension of comparison is error-tolerance, for both manufacturing or hard defects and soft defects. A write-through cache can function with either hard or soft single-bit errors, if parity is provided. This is because the write-through cache contains no unique dirty data, and reads of data with errors can be turned into cache misses. A write-back cache can not tolerate a single-bit error of any type unless ECC is provided. ECC must usually be computed on at least a 32 bit data word to be economical. For example, single bit detection and correction (but not double detection) ECC requires 6 bits per 32 bit word versus 4 bits per 8 bit byte giving 16 bits per 4 bytes. Thus operations like byte store must first read and ECC-decode a word before being able to write a byte. Moreover, byte parity on a four-byte word would allow four single-bit errors to be corrected by refetching a write-through line in comparison to only one error for an ECC-protected write-back cache word. This is true even though byte parity requires only two-thirds of the overhead of word ECC. Thus write-through caches with parity have better error-tolerance at a smaller cost than write-back caches with ECC.

A fifth dimension of comparison is the write bandwidth into the cache (i.e., the number of cycles required per write). A write-back cache must probe the tag store for a hit before the corresponding data is written. This is because if the write access misses and the victim is dirty, unique dirty data will be lost if the cache line is written before the probe. However, a direct-mapped write-through cache can always write a cache line of data at the same time as probing the address tag for a hit. If the access misses, the line is never dirty and will be replaced anyway so there is no problem. If the data cache is set-associative, the probe must occur before the write whether the cache is write-back or write-through. However, a large and increasing number of first-level data caches are direct-mapped, for reasons discussed in [8, 11]. The two-cycle access of straightforward write-back and set-associative cache implementations (i.e., a probe cycle followed by a write cycle) provides more limited store bandwidth at the input to the cache than a direct-mapped write-through cache, in order to reduce the write bandwidth required on the output side of the cache. In machines that can issue multiple instructions per cycle, the incoming load/store bandwidth of the cache can be a limiting factor to machine performance. Although stores are about half as frequent as loads on average, if each store requires two cycles this will result in a 33% reduction in effective first-level cache bandwidth as compared to a machine that only requires one cycle per store. In the next section we consider ways of reducing write-hits to a single cycle in write-back and set-associative write-through caches.

The sixth dimension of comparison is the ease with which stores and their attendant writes are integrated into the machine pipeline (see Figure 3). In a direct-mapped write-through cache writes can always be performed in the pipestage where loads read the cache. If the access turns out to be a miss the conventional miss-recovery hardware provided for load misses can be used, and the store write cycle is simply repeated. However, a simple write-back or set-associative write-through cache can require two cycles of cache access per store: the first cycle probes the cache tags, and the second sets the appropriate dirty bits and writes the data. This will require interlocks when loads immediately follow stores, since the stores would be accessing the data section at the same time as the next (load) instruction is accessing the data section of the cache (i.e., without interlocks the WB pipestage of the store would be at the same time as the MEM pipestage of the load.) Note that if load latency weren't important, loads could delay their data access until WB after hit or miss were already known. Then stores and loads would access the cache with the same timing and could be issued one per cycle in any order. However, since load latency is of critical importance in machine design, this is not a viable option. Although in

Figure 3 stores into a write-through cache would commit a pipestage earlier than loads or other operations (which commit in WB), the cache line written by the store can be flushed a pipestage after its write without adverse consequences. This allows exceptions to be handled precisely. Similarly, data going into the write buffer in the MEM pipestage of Figure 3 can be aged one cycle until the instruction is known to have completed without exception. Table 2 summarizes the advantages and disadvantages of write-back and write-through caches.

pipe-stage	function	load timing	store instruction timing write-through$	write-back*
IF	instruction fetch			
RF	register fetch			
ALU	address calculation			
MEM	cache access	read data read tags	write data read tags	read tags
WB	write register file			write data if tags hit

$ Also assumes direct-mapped.
* This also applies for set-associative write-through caches.

Figure 3: Direct-mapped write-through and write-back pipelines

feature	write-through	write-back
traffic	- more	+ less
additional buffers	- write buffer needed	- dirty victim buffer needed
ability to handle bursty writes	- write buffer can overflow	+ OK unless writes miss with dirty victims
single bit soft or hard error safe	+ with parity	- only with ECC
pipelining	+ same as loads if direct-mapped	- doesn't match
cycles required per write	+ 1	- 1 to 2 (incl. probe)

Table 2: Advantages and disadvantages of write-through and write-back caches

3.1. Increasing Write-Back Cache Bandwidth

One observation to make about probe-before-write is that almost all probes will hit in the cache. If separate address lines are provided going to the tag and data portions of the cache, the bandwidth for a series of write hits can be doubled (see Figure 4) by using a *last-write register*. During the time usually used for probing the tags and reading the data section on a read (i.e., the MEM pipestage of Figure 3), we will probe the tags with the current write address and write the data for the last write. As long as the probe for the last write succeeded, and there were no read misses since the last write probe, we know the line for the last write will still be in the cache. This organization can provide double the write bandwidth going into a write-back or set-associative write-through cache as compared to an organization which has common tag and data address lines. This delayed write method is used in the VAX 8800 [6] (albeit in a write-through cache for other reasons).

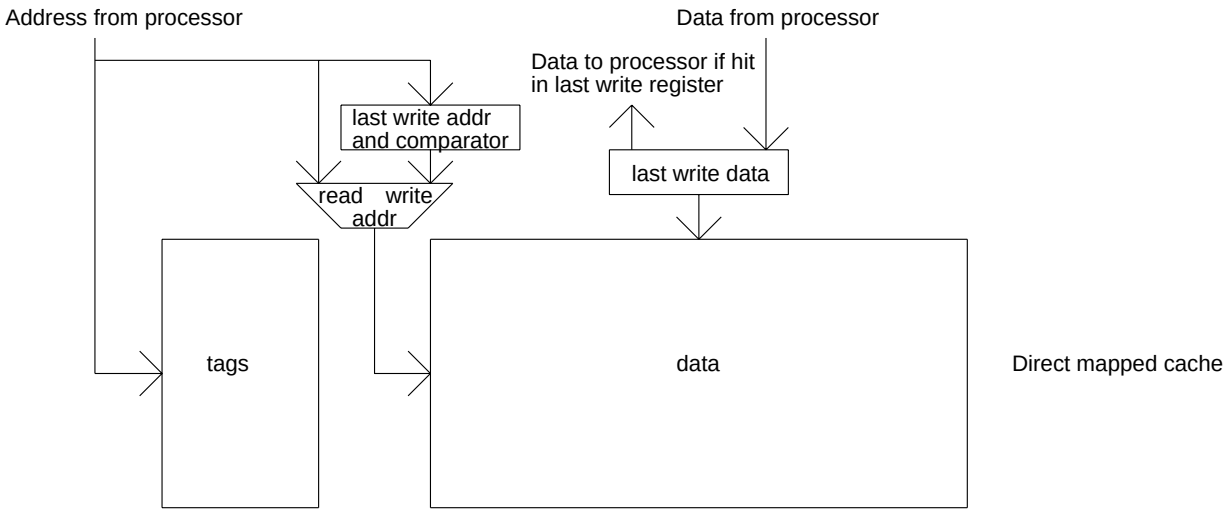

Figure 4: Delayed write method for write-back caches

Several complications result with this method. First, the delayed write address register must also have a comparator so that if a read for the delayed write address occurs before it is written into the cache it can be supplied from the delayed write register. Also, if the write-back cache is on-chip, providing two sets of address lines is relatively easy. However, if the cache is off-chip, having two sets of address lines will require additional pins which are invariably a scarce resource. Finally, if the line size is larger than the width of the cache RAMs, the line dirty bit must be associated with the tag. This means that the write can only be performed in one cycle if the line is already dirty, otherwise the tag dirty bit will need to be written at the same time as the data. Luckily, for large caches most writes access already dirty lines (e.g., Figure 2), so this is not too much of a problem.

3.2. Reducing Write-Through Cache Traffic

The primary problem with write-through caches is their higher write traffic as compared to write-back caches. One way to reduce this traffic is to use a *coalescing write buffer*, where writes to addresses already in the write buffer are combined.

Figure 5 shows the simulation results for an 8-entry coalescing write buffer. Each write buffer entry is a cache line (16B) wide. The data presented are the results of the six benchmarks averaged together. Simulations were performed where the write buffer emptied out an entry every *n* cycles, with *n* varying from 0 to 48 cycles. In practice the number of cycles between retirement of write buffer entries will depend on intervening cache miss service and other system factors. Since cache miss service effectively stops processor execution in many processors, cache misses were ignored in Figure 5. This allows a fixed time between writes to be used as a reasonable model of the write buffer operation. If dirty write buffer entries are written back quickly, they do not stay in the write buffer for many cycles and hence relatively little merging takes place. For example, if write buffer entries are retired every 5 cycles, the write traffic is reduced by only 10%. The only way that a significant number of writes are merged (e.g., 50% or more) is if the write buffer is almost always full. But in this case stores almost always stall because no write buffer entries are available. For example, to attain a write traffic reduction of 50%, writes must be retired no more frequently than every 38 cycles, resulting in a CPI burden of 7! Since much of current computer research is focused on achieving machines with CPIs of less than one, write buffer stalls should be well under 0.1 CPI. This means that only a small percentage of writes (e.g., less than 20%) can be merged with simple coalescing write buffers. The extra traffic resulting from this lack of coalescing wastes cache bandwidth that could otherwise be used for prefetching or other uses.

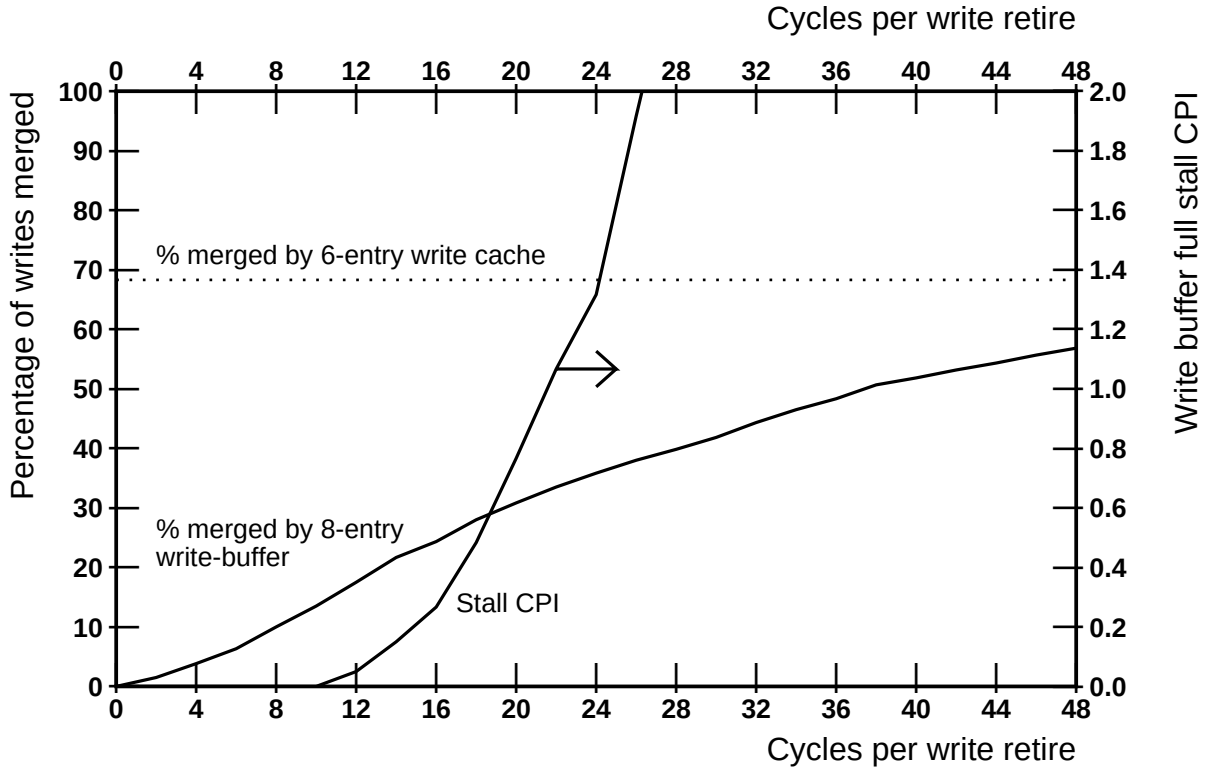

Figure 5: Coalescing write buffer merges vs. CPI

Instead of having writes enter and leave the write buffer as soon as possible, we can add a *write cache* in front of the write buffer and behind the data cache. A write cache is a small fully-associative cache (see Figure 6). With a small number of entries we can try to coalesce the majority of writes and decrease the write traffic exiting the chip. When a write misses in the write cache, the LRU entry is transferred to the write buffer to make room for the current write. In actual implementation, the write cache can be merged with a coalescing write buffer. Here a write buffer of *m* entries would only empty an entry if it has more than *n* valid entries, where *n* is the number of entries conceptually in the write cache (with $m > n$). A write cache can also be implemented with the additional functionality of a victim cache [10], in which case not all entries in the small fully-associative cache would be dirty.

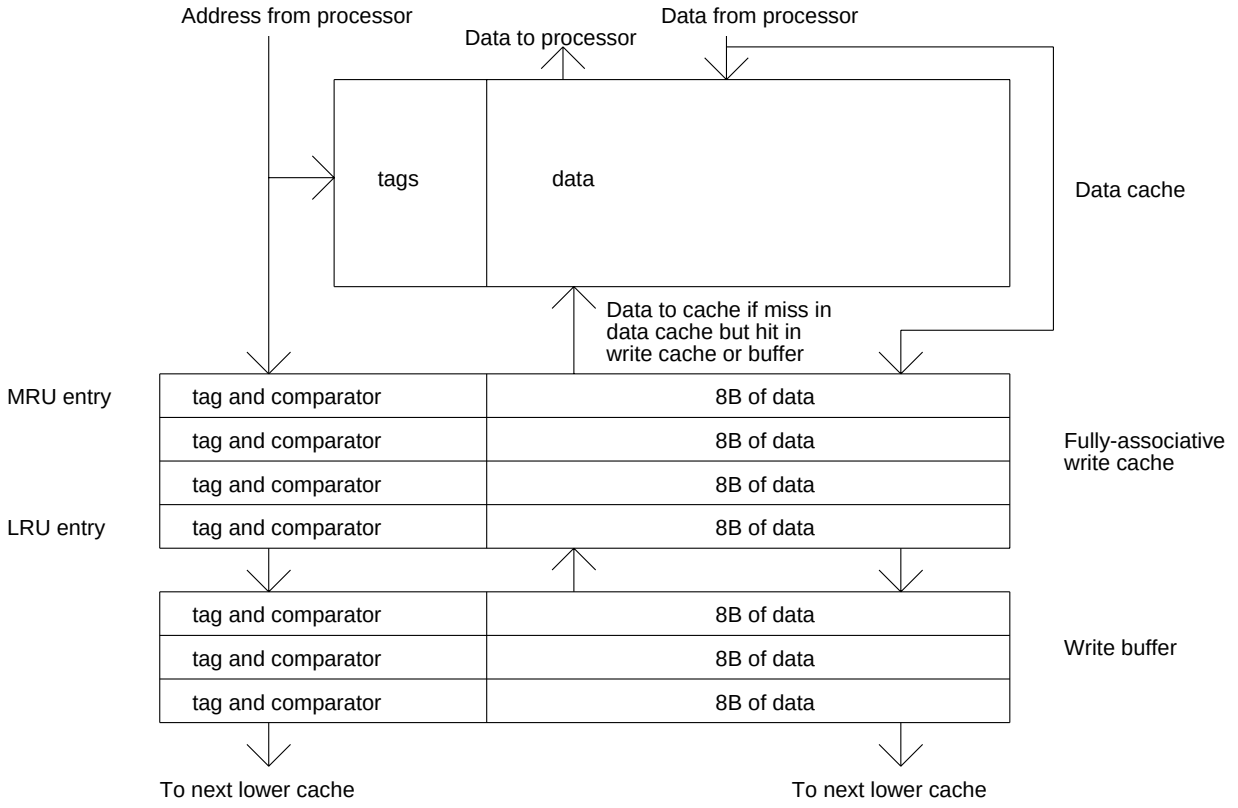

Figure 6: Write cache organization

Figure 7 gives the number of writes removed by a write cache with varying numbers of 8B lines. (8B was chosen as the write cache line size since no writes larger than 8B exist in most architectures, and write paths leaving chips are often 8B.) A write cache of only five 8B lines can eliminate 50% of the writes for most programs. Two notable exceptions to this are *linpack* and *liver*. Because these programs sequentially travel through large arrays, even a write-back cache of modest size (less than 32KB) removes very few writes. In order to get a better idea of how write caches compare with write-back caches, the write traffic reduction of a write cache is given relative to a 4KB write-back cache in Figure 8. In Figure 8 a write cache of only four 8B entries removes over 50% of the writes removed by a 4KB write-back cache on all of the benchmarks except *met*. Another interesting result is that a write cache with eight or more 8B entries actually outperforms a 4KB direct-mapped write-back cache on *liver*. This is because

mapping conflicts within the write reference stream prevent a direct-mapped write-back cache from being as effective at removing write traffic as the fully-associative write cache.

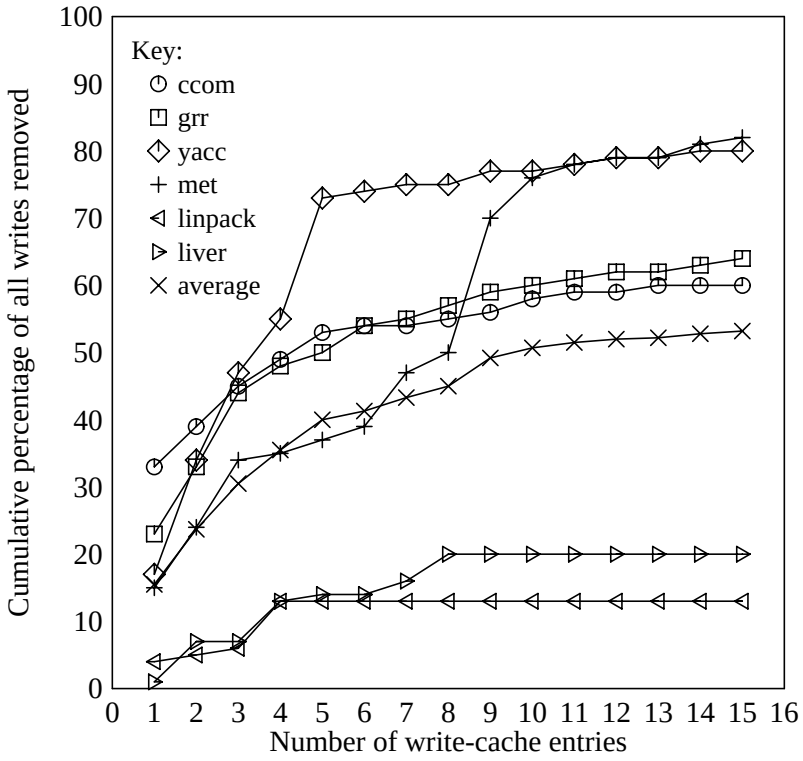

Figure 7: Write cache absolute traffic reduction

Figures 7 and 8 also give the average traffic reduction of write caches in absolute terms and relative to a write-back cache. The two most interesting points on these curves are probably a five-entry write cache, since it seems to be at the knee of the traffic reduction curve, and a one-entry write cache, since it is the simplest to implement. The five-entry write cache can remove 40% of all writes, or 63% of those removed by a 4KB write-back cache. The single-entry write cache can remove 16% of all writes on average, which is 21% of the writes removed by a write-back cache.

Of course the relative traffic reduction of a write cache varies as the size of the write-back cache used in the comparison varies (see Figure 9). Compared to a 1KB write-back cache, a five-entry write cache removes 72% of the write traffic but compared to a 32KB write-back cache it only removes 49% of the write traffic. This change is surprisingly small considering the 32:1 ratio in write-back cache size, and is due to the write cache's good absolute traffic reduction. The reduction in write cache relative effectiveness is fairly uniform as the write-back cache size used for comparison increases in size.

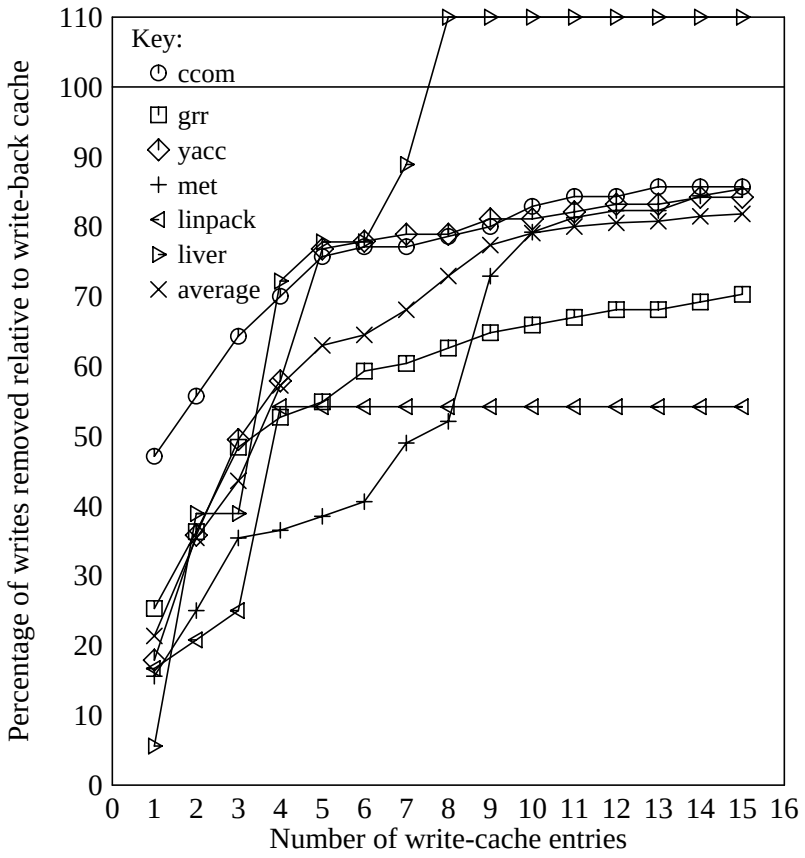

Figure 8: Write cache traffic reduction relative to a 4KB write-back cache

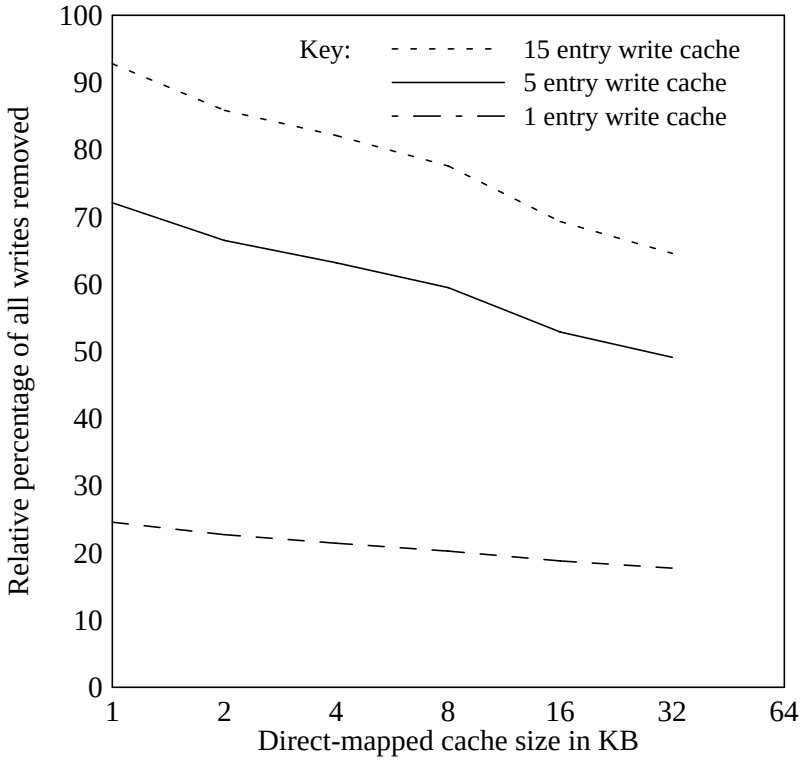

Figure 9: Relative traffic reduction of a write cache vs. write-back cache size

3.3. Summary: When to Choose Write-Back or Write-Through

From the previous sections, it is clear that both high performance write-back and write-through caches require a fair amount of additional support hardware and complexity, such as dirty bits, dirty victim buffers, delayed write register, write caches, and write buffers. In fact the hardware requirements for high performance write-back and write-through caches are surprisingly similar. For example, a high performance write-back cache requires a dirty victim register, while a write-through cache requires a corresponding write-buffer (see Table 3). Similarly, a high performance write-back cache requires a delayed write register to improve write bandwidth into the first-level cache, while a write-through cache requires a write cache to significantly reduce the bandwidth requirements of store traffic exiting the cache. (Set-associative write-through caches would require both a delayed-write register and a write cache.) In each of these last two items the write-through cache required a buffer with 3 to 5 entries, while the write-back cache only required a single register. However, the write-back cache requires a dirty bit on every cache line, while the write-through cache does not require any dirty bits at all. The extra real estate needed by the write-back cache's dirty bits offsets the extra hardware required to provide buffers instead of single registers for the write-through cache.

feature	write-back	write-through
exit traffic buffer	dirty victim register	write buffer
bandwidth improvement	delayed write register	write cache
other	cache line dirty bits	

Table 3: Hardware requirements for high performance write-back and write-through caches

Once these improvements are made to the write-back and write-through caches, the only remaining differences between them are their fault tolerance and their remaining write traffic. For a 4KB direct-mapped cache, a five-entry write cache reduces the write traffic by 40%. If a write-back cache is used instead an additional 18% reduction in write traffic can be obtained on average over the six benchmarks. Thus for small and moderate size caches further reductions of this magnitude will probably not be worth the hardware overhead of implementing ECC for fault tolerance on the write-back cache. Since the reduction in write traffic provided by a write-back cache increases as its size increases, write-back caches become more attractive in comparison to write-through caches with write caches for very large on-chip caches. For example, a 32KB write-back data cache can remove an additional 32% of the write traffic over a write-through cache with a five-entry write cache. This reduces by half the amount of remaining write traffic as compared to a write-through cache.

4. Write Misses: Fetch-on-Write vs. Write-Validate vs. Write-Around vs. Write-Invalidate

The policy used on a write that misses in the cache (i.e., "write miss") can significantly affect the total amount of cache refill traffic, as well as the amount of time spent waiting during cache misses. The number of cache misses due to writes varies dramatically depending on the benchmark used. Figure 10 shows the percentage of misses that are due to writes for various cache sizes with 16B lines. Figure 11 shows the percentage of misses that are due to writes for an 8KB cache with various line sizes. On average over all the cache configurations, write misses account for about one-third of all cache misses. Since loads outnumber stores in these benchmarks by roughly 2.4:1 (see Table 1), this means that stores are about as likely to cause a miss as loads.

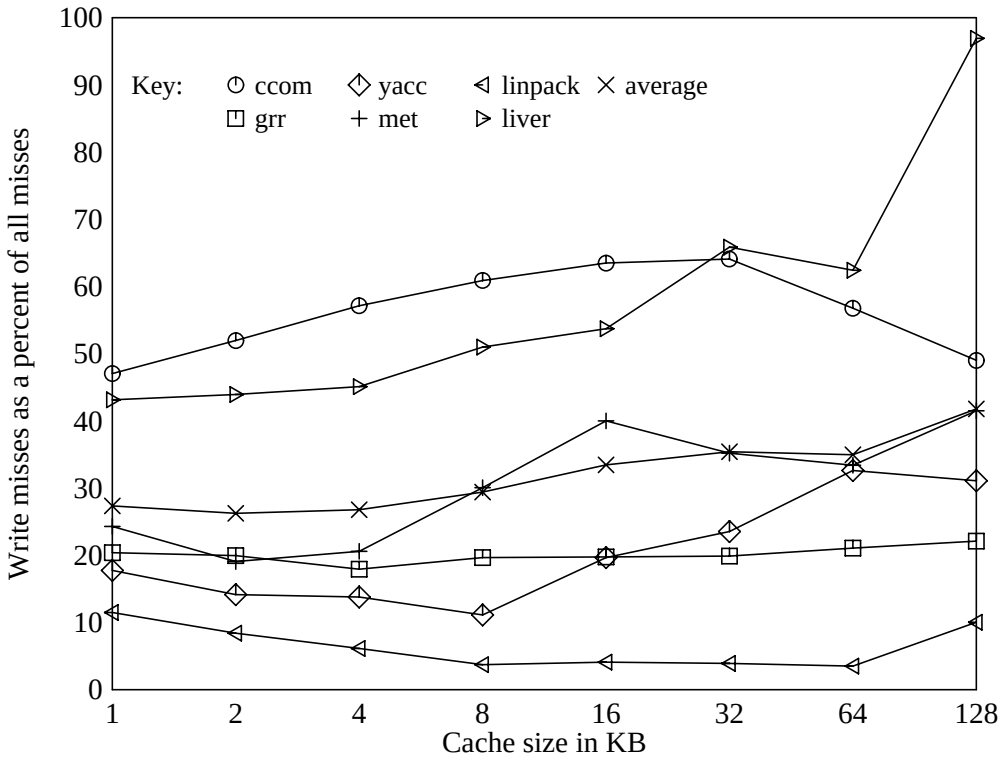

Figure 10: Write misses as a percent of all misses vs. cache size for 16B lines

Unfortunately, cache operation on a write miss is an even more neglected and confused subject in the literature than cache operation on write hits. There are four combinations of three policies from which to choose (see Figure 12).

In systems implementing a *fetch-on-write* policy, on a write miss the line containing the write address is fetched. In systems implementing a *write-allocate* policy, the address written to by the write miss is allocated in the cache. In systems implementing a *write-invalidate* policy, the line written to by the write miss is simply marked invalid. Although write-invalidate may initially sound like a contrived policy, it can actually be quite useful in practice. If a direct-mapped write-through cache is being used, the data can be written concurrently with the tag

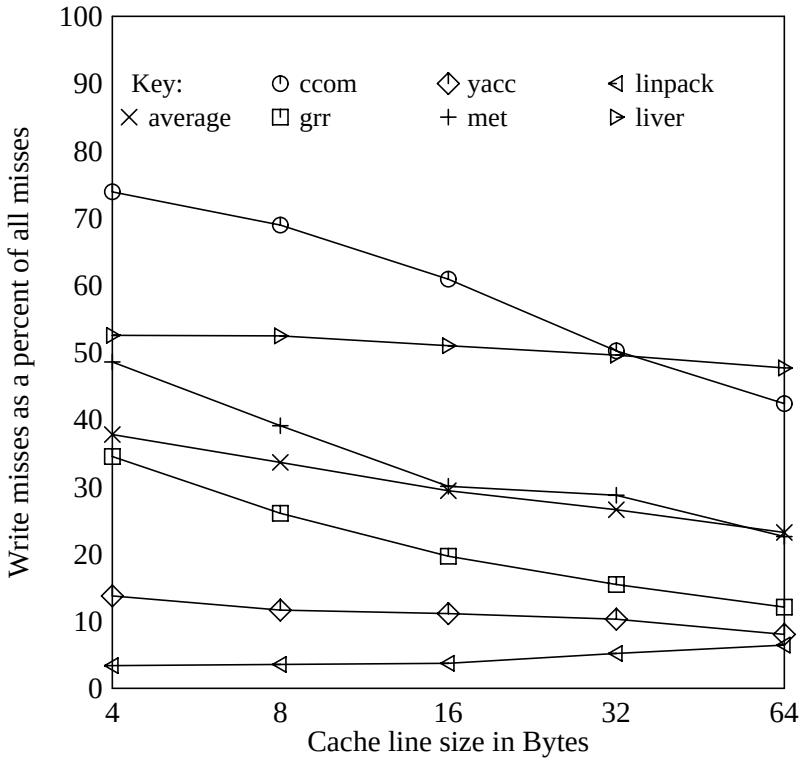

Figure 11: Write misses as a percent of all misses vs. line size for 8KB caches

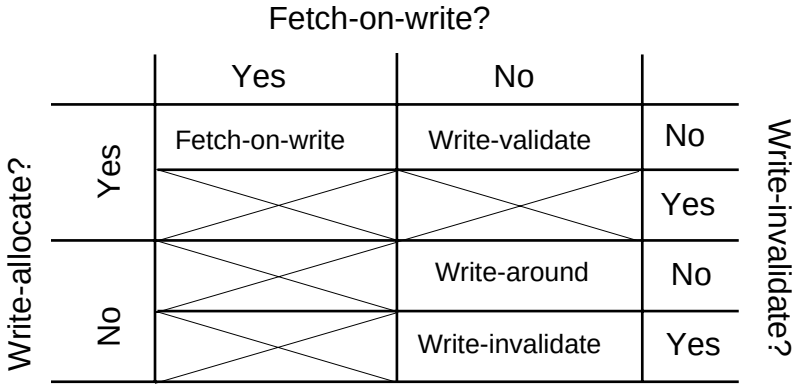

Figure 12: Write miss alternatives

check. If the tag does not match, the data portion of the line has been corrupted (i.e., assuming the line size is larger than the amount of data being written, the data is a mixture of information from two cache lines). In this case the line can simply be marked invalid, since the data is being written to a lower level in the memory hierarchy anyway. This invalidation can usually be done in a single cycle, or sometimes even in parallel with subsequent cache accesses, and so it is much faster than fetching the correct contents of the cache line being written.

The combinations of fetch-on-write and no-write-allocate or write-invalidate are not useful, since the old data at the write miss address is fetched but is discarded or invalidated instead of being written into the cache. Therefore, fetch-on-write has been used to imply write-allocate in

the literature. Similarly, combinations of write-allocate and write-invalidate are not useful since the line is allocated but is marked invalid. If the old data at the write miss address is not fetched (i.e., no-fetch-on-write), three options are possible. We call the combination of no-fetch-on-write, write-allocate, and no-write-invalidate *write-validate*. With write-validate, the line containing the write is not fetched. The data is written into a cache line with valid bits turned off for all but the data which is being written. We call the combination of no-fetch-on-write, no-write-allocate, and no-write-invalidate *write-around*, since writes do not go into the cache but go around it to the next lower level in the memory hierarchy, leaving the old contents of the line in place. The only useful combination with write-invalidate has no-fetch-on-write and no-write-allocate, so this combination can simply be called write-invalidate.

We call write misses that that do not result in any data being fetched with a write-validate, write-around, or write-invalidate policy *eliminated misses*. For example, with write-validate if the invalid part of a line is never read, the fetch of the data (and the attendant stalling of the processor) is eliminated. A miss is counted only if the invalid portion of a line resulting from the write-validate strategy is read without first being written or the line being replaced. Similarly, with write-invalidate a miss is counted only if the line being written or the old contents of the cache line are read before another address mapping to the same cache line misses. Finally, with write-around, a miss is counted only if the data being written is read before any other data which maps to the same cache line is read. This terminology neglects the time required to set the valid bits on an eliminated miss. However, if maintenance of the valid bits cannot be done in parallel with other operations, it typically takes at most a cycle, which is insignificant compared to cache miss penalties.

The write miss policy used is sometimes dependent on the write hit policy chosen. Write-around and write-invalidate (i.e., policies with no-write-allocate) are only useful with write-through caches, since writes are not entered into the cache. Fetch-on-write and write-validate can be used with either write-through or write-back caching.

Write-validate requires the addition of valid bits within a cache line (i.e., subblocking). Valid bits are usually added on a word basis, so that words can be written and the remainder of the line marked invalid. In systems that allow byte writes or unaligned word writes, byte valid bits would be required for a pure write-validate strategy. However, the addition of byte valid bits is a significant overhead (one bit per byte, or 12.5%) in comparison to a valid bit per word (3.1%). Thus, in practice machines with byte writes that have write-validate capability for aligned word and double-word writes would probably provide fetch-on-write for byte writes.

Write-validate also requires that lower levels in the memory system support writes of partial cache lines. Partial line writes are not difficult to implement; many systems already provide this for uncached writes.

In multiprocessor systems with coherent caches, if write-validate is used on a write-back cache all write misses should write through. If this is not done, the remainder of the system will not know that the processor has dirty data for that cache line in its cache.

The choice of write miss policy can make a significant difference in the performance of certain operations. For example, consider copying a block of information. If fetch-on-write is used, each write of the destination must hit in the cache. In other words, the original contents of the

target of the copy will be fetched even though they are never used and are only overwritten with write data. This will reduce the bandwidth of the copy by wasting fetch bandwidth. Given a total bandwidth available for reads and writes, a fetch-on-write strategy would have only two-thirds of the performance on large block copies as a no-fetch-on-write policy since half of the items fetched would be discarded.

Some architectures have added instructions to allocate a cache line in cases where programmer directives specify or the compiler can guarantee that the entire cache line will be written and the old contents of the corresponding memory locations will not be read [12, 9, 4]. These instructions are limited to situations where new data spaces are being allocated, such as a new activation record on a process stack, or a new output buffer is obtained from the operating system. Unfortunately there are a number of problems that prevent broader application of software cache line allocation:

1. The entire cache line must be known to be written at compile time, or if some of the line is not written its old contents must not need to be saved. (In contrast, write-validate can allow partial lines to written, and is not subject to optimization limitations such as incomplete alias information, etc.)

2. Cache line sizes vary from implementation to implementation, limiting object code using these instructions to the machines with cache line sizes equal to or smaller than that assumed in the allocate instructions.

3. Context switches after a line has been allocated and partially written but before it has been completely written result in dirty and incorrect cache lines. (One way around this would be to add valid bits to each write quantum in the line, but this provides the hardware support needed for write-validate).

4. There is extra instruction execution overhead for the cache allocation instructions.

Thus, the use of cache line allocation instructions is limited to situations such as new data allocation and buffer copies. Write-validate can provide better performance than cache line allocation instructions since it is also applicable in cases where only part of a line is being written or it is not possible to guarantee that an entire line is written at compile time. Write-validate works for machines with various line sizes, and does not add instruction execution overhead to the program. Finally, since write-validate has sub-block valid bits there are no problems with cache lines being left in an incorrect state on context switches.

Figure 13 shows the reduction in write misses (i.e., not including misses on reads) for write-validate, write-around, and write-invalidate for caches with 16B lines. Note that fetch-on-write fetches a cache line on every write miss, so it corresponds to the X axis in Figure 13. In general write-validate performs the best, averaging more than a 90% reduction in write misses. The two no-write-allocate strategies, write-around and write-invalidate, have an average reduction in write misses of 40-65% and 30-50% respectively. Write-around has a greater than 100% reduction in write misses for 32KB and 64KB caches when running *liver*. *liver* is a synthetic benchmark made from a series of loop kernels, and the results of loop kernels are not read by successive kernels. However, successive loop kernels read the original matrices again. The range of cache sizes from 32KB to 64KB is big enough to hold the initial inputs, but not the results too. Since write-around does not place the results in the cache but keeps the old contents of the cache line unchanged, it can also result in fewer read misses since the initial data is not replaced with write data or invalidated.

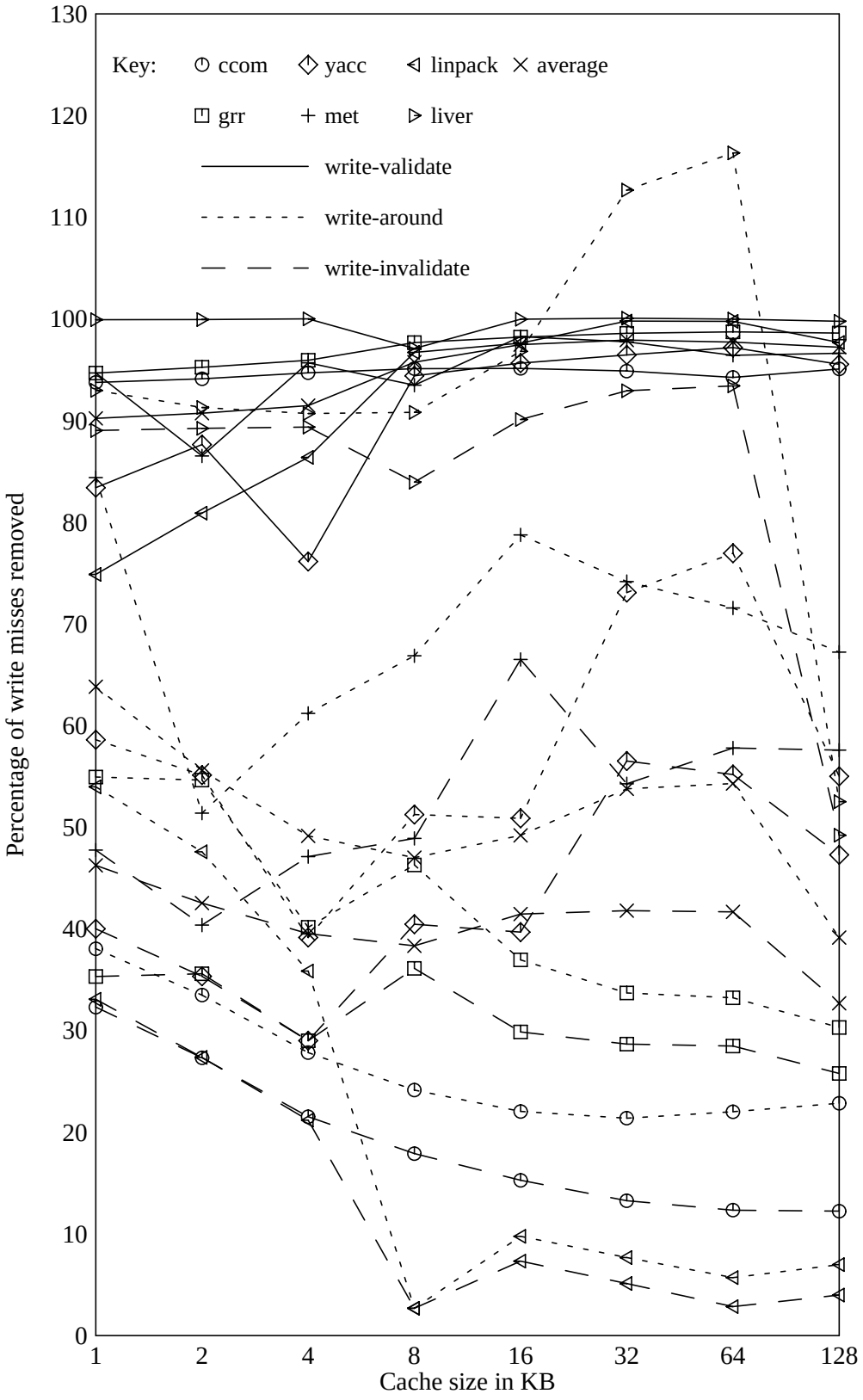

Figure 13: Write miss rate reductions of three write strategies for 16B lines

Figure 14 shows the reduction in data cache misses (including both read and write) for write-validate, write-around, and write-invalidate for caches with 16B lines. (This graph is basically Figure 13 multiplied by Figure 10.) *ccom* and *liver* benefit the most from a write-validate policy. This can be explained as follows. Many of the operations in *ccom* and *liver* are similar to copies: data is read but other data is written. For example, array operations of the form "for j := 1 to 1000 do A[j] := B[j] + C[j]" only write data which is never read before being written. Similarly, write-validate would be useful for a compiler if it has a number of sequential passes, each one reading the data structure written by the last pass and writing a different one. The other programs have more read-modify-write behavior. The best example of this is *linpack*. The inner loop of *linpack*, saxpy, loads a matrix row and adds to it another row multiplied by a scalar. The result of this computation is placed into the old row. Here write-validate would be of very little benefit since almost all writes are preceded by reads of the data anyway. On average over the six programs write-validate reduced the total number of data cache misses (over both read and write) by 31% for an 8KB data cache with 16B line size.

Write-around performs well when the data being written by the processor is not read by it soon or ever. This is the situation in *liver* with a 32KB or 64KB cache, the only benchmark that performs better with write-around than write-validate. In general, however, most programs are more likely to read what they have just written than they are to re-read the old contents of a cache line. For all other cases the performance of write-around is worse than that of write-validate.

Write-invalidate does not show as much improvement over fetch-on-write as the other two strategies, but it still performs surprisingly well. *livermore* has about a 40% reduction in misses, and the six benchmarks on average have a 10-20% total reduction in misses compared to fetch-on-write. Moreover, write-invalidate is very simple to implement. In a write-through cache using write-invalidate the data can be written at the same time the tags are probed. If the access misses, the line has been corrupted so it can be simply marked invalid, often without inserting any machine stall cycles.

Figure 15 shows the reduction in write misses (i.e., not including misses on reads) for write-validate, write-around, and write-invalidate for 8KB caches with various line sizes. Since fetch-on-write fetches a cache line on every write miss, it corresponds to the X axis in Figure 15. Write-validate, write-around, and write-invalidate have the highest benefit for small lines. If the line size is the same as the item being written, any old data fetched by fetch-on-write is merely discarded when the write occurs. As the line size gets larger, the odds that some old data on the line will be needed increases, so the advantage of write-validate decreases. The miss rate reduction of write-around also decreases with increasing line size for similar reasons. The performance advantage of write-invalidate decreases with increasing line sizes because more information is being thrown away. Again write-validate performs the best, averaging more than a 90% reduction in write misses except at the longest line sizes. The two no-write-allocate strategies, write-around and write-invalidate, have an average reduction in write misses of 40-70% and 35-50% respectively.

Figure 16 shows the reduction in total misses for write-validate, write-around, and write-invalidate for 8KB caches. (This graph is basically Figure 15 multiplied by Figure 11.) Again write-around generally performs worse than write-validate, because most programs are more likely to read the data that was just written than the old contents of the cache line. Both write-

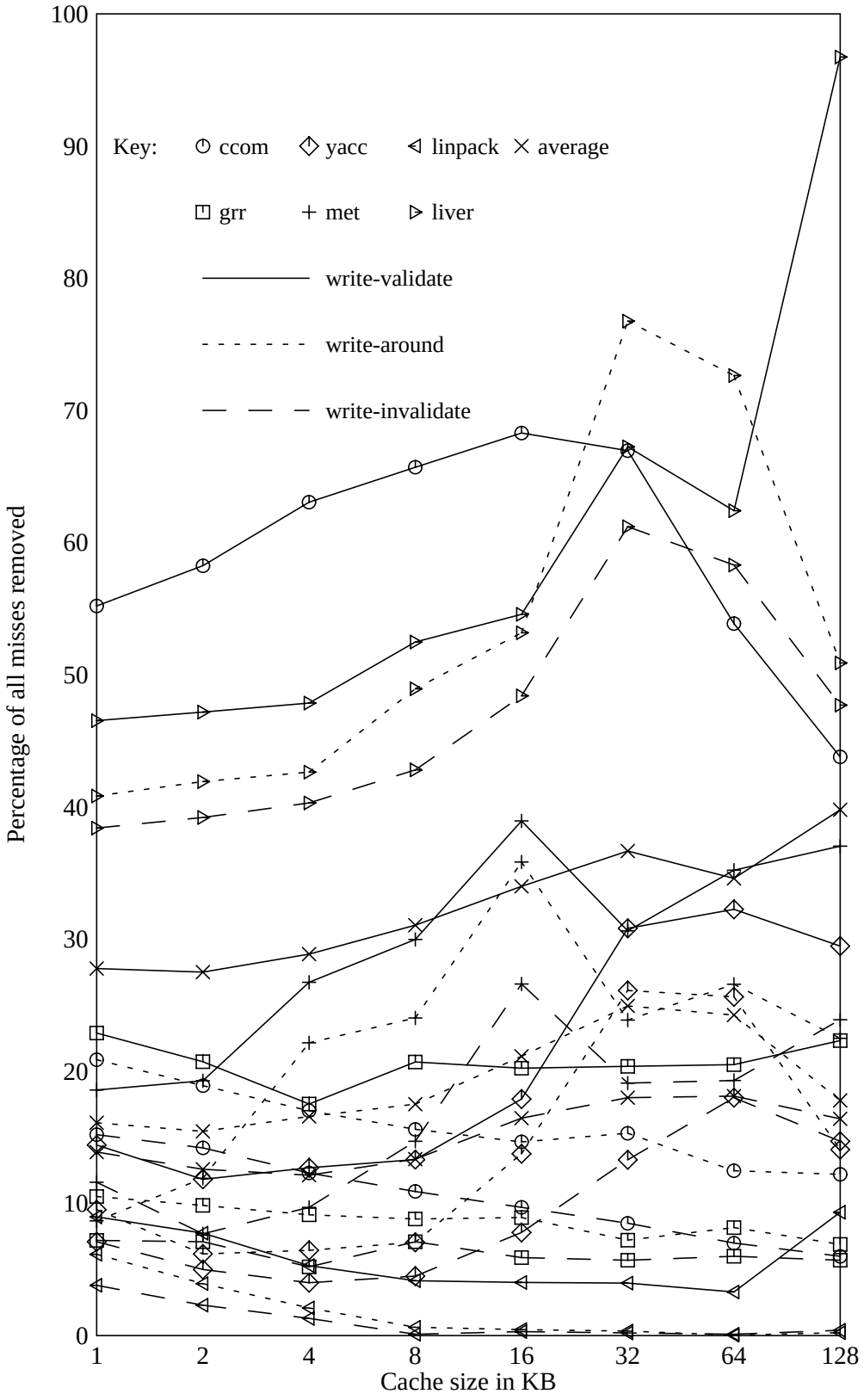

Figure 14: Total miss rate reductions of three write strategies for 16B lines

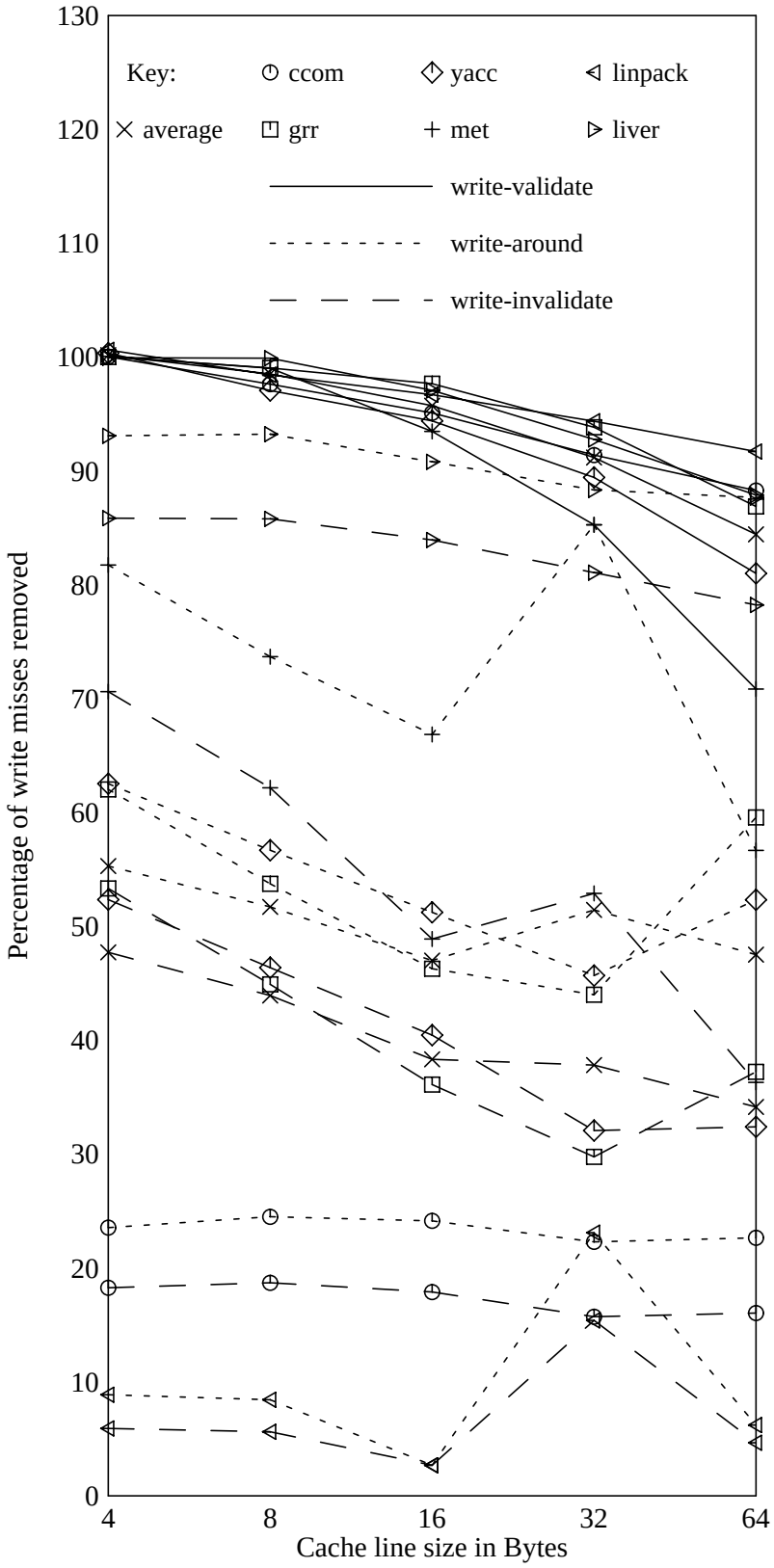

Figure 15: Write miss rate reductions of three write strategies for 8KB caches

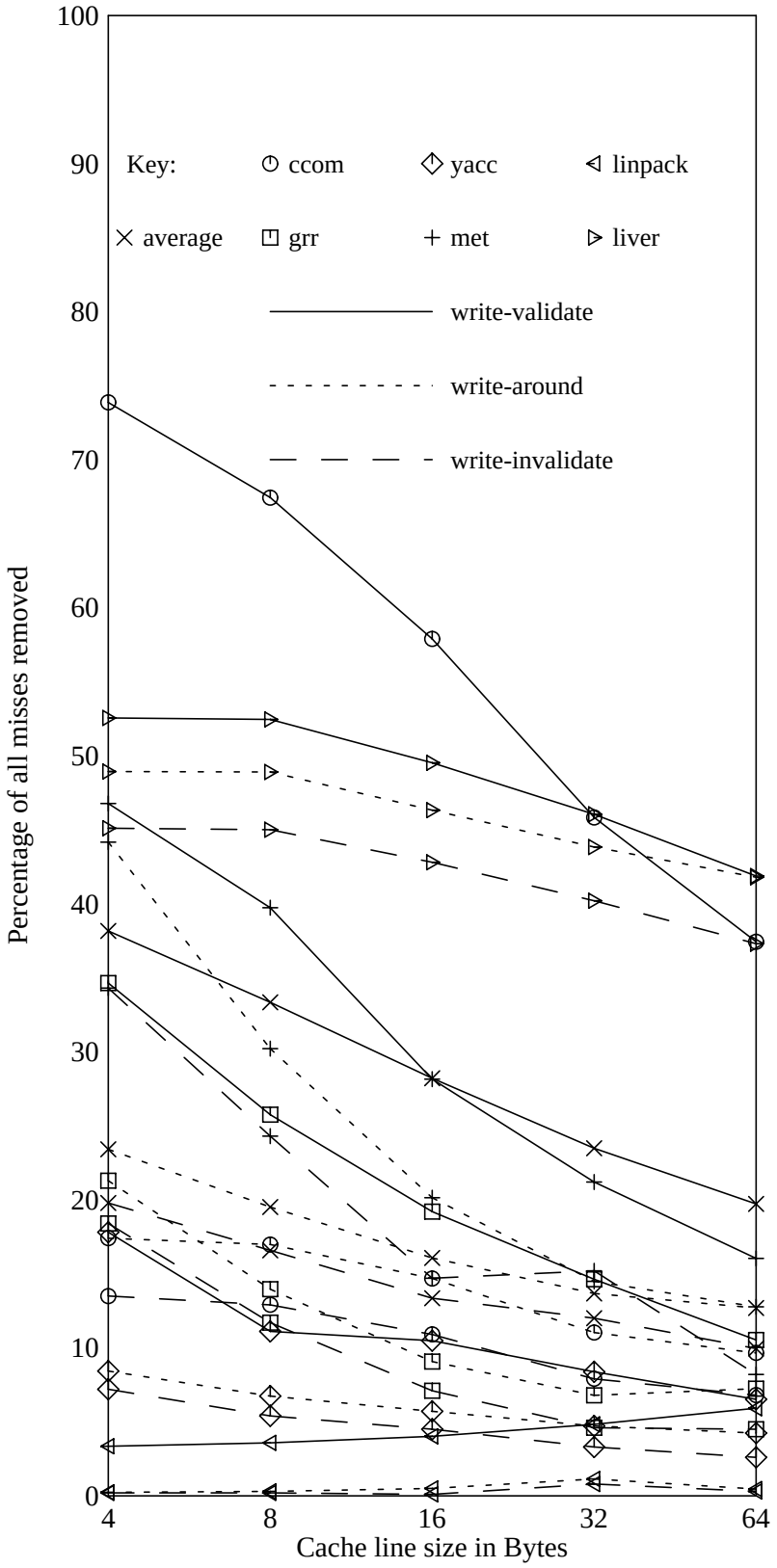

Figure 16: Total miss rate reduction of three write strategies for 8KB caches

validate and write-around perform better than write-invalidate, but again write-invalidate performs surprisingly well. The ratio of miss rate reduction of write-validate to write-around decreases as the line size increases since write-validate invalidates an increasing number of bytes while write-around leaves all the bytes on the line valid.

We can generate a partial order of the relative miss traffic between these four write-miss policy combinations (see Figure 17). Fetch-on-write always has the most lines fetched, since it fetches a line on every miss. Write-invalidate avoids misses in the case where neither the line containing the data being written nor the old contents of the cache line are read before some other line mapping to the same location in the cache is read. This saves some misses over fetch-on-write. Write-around and write-validate always have fewer misses than write-invalidate. Write-around avoids fetching data in the same cases as write-invalidate, as well as cases where the old contents of the cache line are accessed next. Write-validate avoids fetching data in the same cases as write-invalidate, as well as cases where the data just written is accessed next. Usually the data just written (i.e., write-validate) is more useful than the old contents of the cache line (i.e., write-around), but this is not always the case.

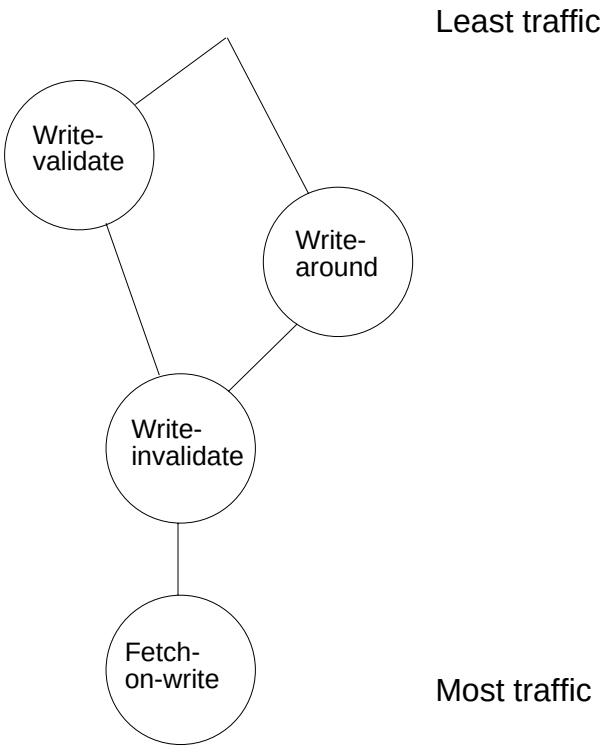

Figure 17: Relative order of fetch traffic for write miss alternatives

5. Traffic Out the Back of the Cache

In this section we measure the traffic at the back end of a cache with two metrics. Section 5.1 considers traffic on a transaction basis, where a transaction is a cache line fetch, write-back, or data being written through. Section 5.2 considers traffic measured in bytes.

Care must be taken when measuring cache write-back traffic. For small benchmarks and large write-back caches, at the end of a simulation almost all writes may still be in the cache. This is because very few writes may have come out the back of the cache due to cache line replacement. We call this situation a case where *cold stop* effects are important. For example, *liver* running with a 128KB data cache with 16B lines only creates 454 dirty victims during its execution, but 6014 dirty lines remain in the cache. Flushing dirty lines from the cache will result in almost as much traffic as all of the 6541 read misses for *liver* with these cache parameters. Similarly, when *yacc* runs with these cache parameters, 22% of the lines written during program execution still reside in the cache.

In this section in cases where cold-stop effects are significant, it is assumed that the data cache is flushed of dirty cache lines after program execution. The flush traffic is added to the write-back traffic from program execution.

Another way to account for cold stop behavior is to start the simulation with a statistically appropriate number of dirty blocks in the cache [5]. Since some benchmarks leave a higher percentage of dirty lines in the cache than others, it is probably best if the same program is run twice. The first execution will give the final percentage of dirty lines remaining. The second execution can start with the percentage of dirty lines left by the first execution. Note that the initially dirty lines must be marked with non-matching but valid tags to generate write-back traffic.

5.1. Traffic Measured in Transactions

We can combine the data from the write-hit studies in Section 3 and the data from the write-miss studies in Section 4 with read miss statistics to get an overall picture of components of the traffic out the back of a data cache. Figure 18 shows the number of transactions out the back of a data cache vs. cache size. For the read miss and write miss cases these transactions move an entire cache line worth of data. The additional transactions from write-through and write-back dirty victim traffic may be smaller than a cache line in some systems. However, for the purpose of these charts only transactions and not bytes are counted. The large drop in miss rate between 64KB and 128KB is due to the benchmarks *liver* and *yacc* fitting in a 128KB cache.

Figure 18 shows that the number of transactions out the back of a data cache varies by less than a factor of two for a write-through cache over a two-decade change in cache size. This is because the traffic is dominated by store traffic. Write caches can be used to reduce the number of transactions to approximately midway between that of a write-through and write-back cache. The write-back cache traffic is composed of three parts: read miss traffic, write miss traffic, and dirty victim traffic. Not all cache victims are dirty, so the write-back cache typically has 40 to 80% additional transactions over the total miss traffic alone. By using write-miss strategies such as write-validate instead of fetch-on-write, a large number of the write misses can be eliminated. This can result in a total traffic for write-back caches that is closer to the total of read miss and write-back traffic than the total of read miss, write miss, and write-back traffic.

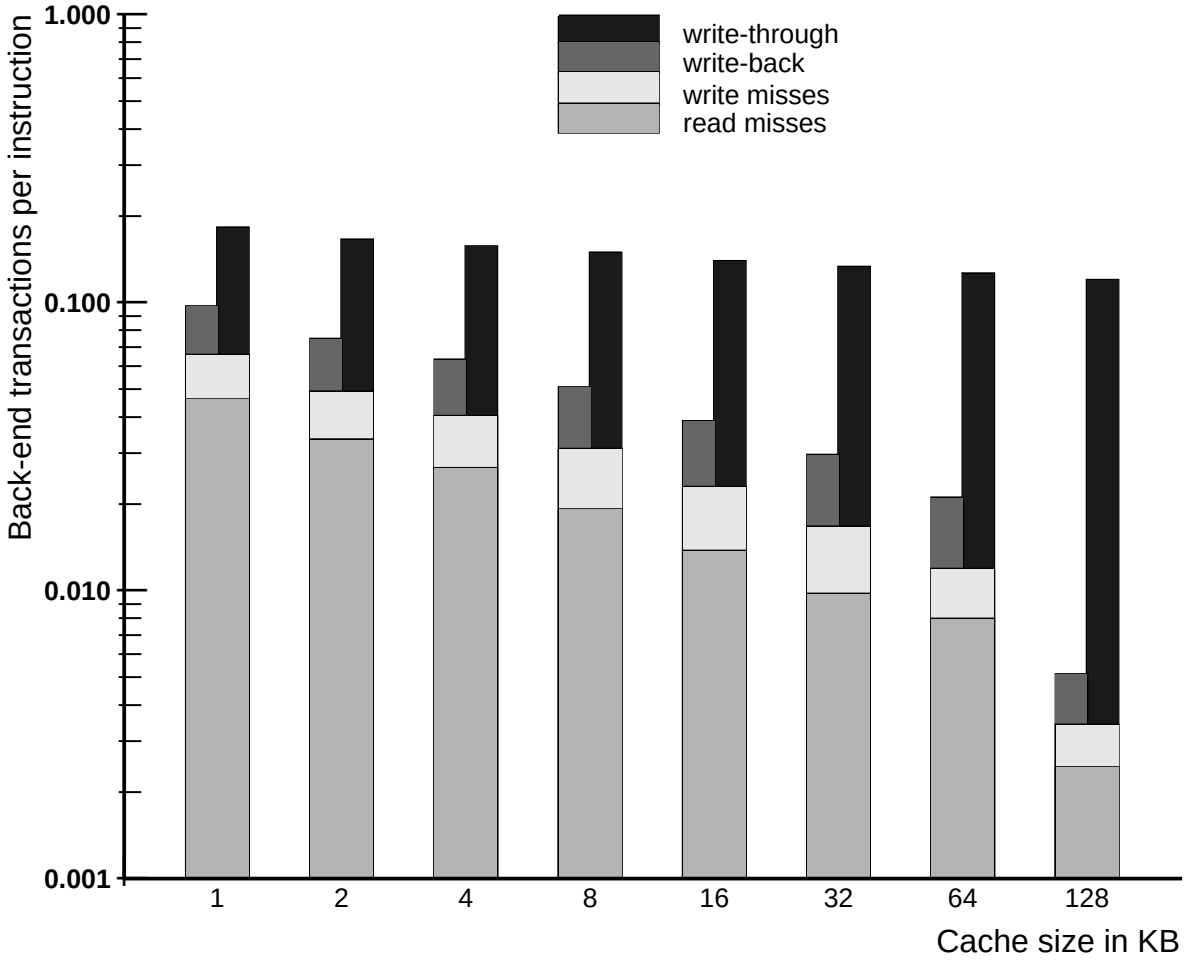

Figure 18: Components of traffic vs. cache size

Figure 19 shows the components of traffic out the back end of a data cache vs. cache line size for an 8KB cache. As the cache line size increases, the number of transactions decreases (although the number of bytes transferred increases). Again, the write-through traffic is dominated by the store traffic, so it varies by less than a factor of 2 over a decade range in cache line size. Write-caches and write-miss strategies other than fetch-on-write will have the same effects as they did in Figure 18.

5.2. Traffic Measured in Bytes

The previous section presented statistics based on the number of read and write transactions at the back of the cache, ignoring the number of bytes with each transaction. However, when implementing actual systems, in order to choose the width of the port from the cache to the next lower level in the memory systems, information on the actual traffic in bytes is more useful. The read traffic in bytes (assuming no partial line fetches) is simply the number of read transactions times the line size. However, on the write side the issues are more complicated, since not all bytes within a line are dirty. This section presents statistics on the percentage of victims dirty, and the percentage of bytes dirty in a victim with dirty bytes for various cache configurations.

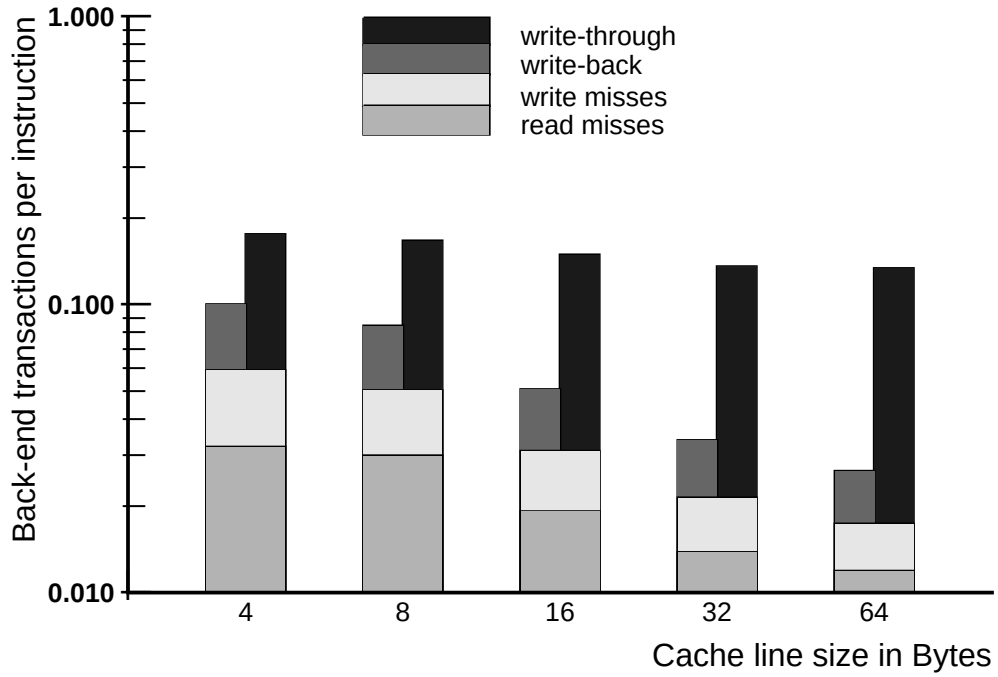

Figure 19: Components of traffic vs. cache line size

Of course this data is limited to write-back caches, since write-through caches do not have dirty victims. This data can be used to answer two basic questions:

- What average write-back bandwidth is needed, relative to the fetch bandwidth?
- Should a write-back write out an entire cache line, or just write out subblocks with dirty bytes? (i.e., are subblock dirty bits useful?)

Figure 20 shows the percentage of data cache victims that have at least one byte dirty vs. cache capacity for 16B lines. The solid lines include only victims from program execution (i.e., cold stop). The dotted lines assume that the cache is flushed after execution. A percentage of these cache lines flushed will be dirty. The weighted average of the cold stop dirty victim percentage and the flushed lines dirty victim percentage gives the dotted points plotted. In general, as the cache size gets larger, the percentage of victims which are dirty increases slightly, although the absolute number of dirty victims decreases, as in Figure 18. The major exceptions to this are cold stop numbers for *liver* above 64KB and *yacc* above 32KB. As discussed previously, for these cache parameters most writes are still in the cache at the end of program execution. The flush stop data for these benchmarks and cache configurations is more in keeping with the statistics from the other programs and the general trendline. On average, about 50% of the victims are dirty, but this percentage varies widely from program to program.

The percentage of bytes dirty in a victim with dirty bytes is given in Figure 21. For small caches the average percentage is about 70%, but it gradually increases with cache size to almost 90%. The numeric benchmarks usually write all the bytes on a cache line if they write any. This is because the numeric benchmarks which were simulated have unit stride. Non-unit stride numeric programs would write a much smaller percentage of their dirty line bytes, especially for long line sizes.

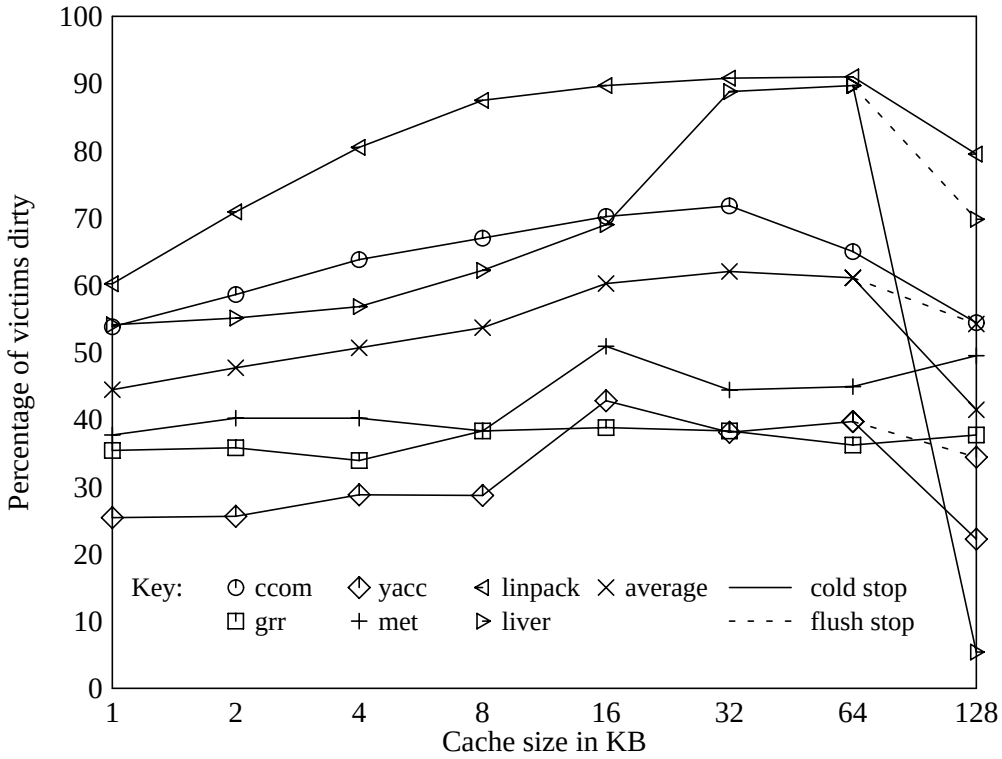

Figure 20: Percent of victims with dirty bytes vs. cache size for 16B lines

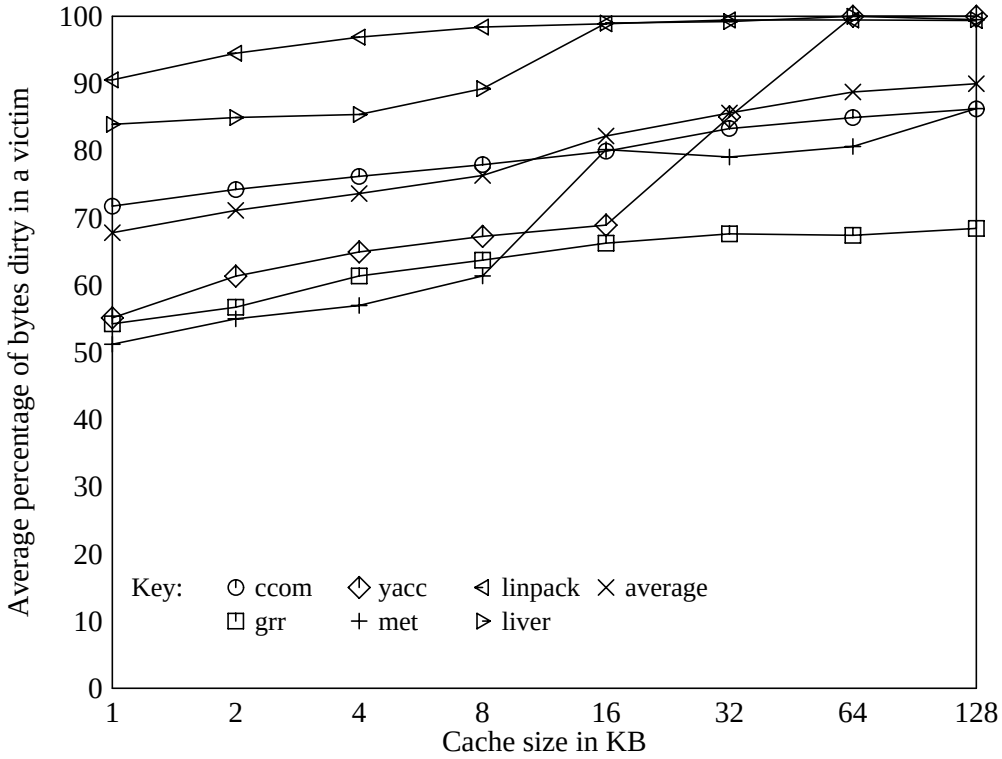

Figure 21: Percent of bytes dirty in a dirty victim vs. cache size for 16B lines

Figure 22 gives the percentage of bytes in a victim line that are dirty, for a range of direct-mapped cache sizes with 16B lines. Note that this data is averaged over all victims, whether clean or dirty. Effectively Figure 22 is the product of Figures 20 and 21, except that Figure 22 uses only flush stop data. Except for anomalies with cache sizes that are large in comparison to the benchmarks (e.g., 128KB), the percentage of dirty bytes per victim gradually increases as the cache size gets larger. This is due to the fact that as a cache gets larger, the hit rate increases. This increases the chances that a write will hit, and therefore write to an already dirty line. This increases the average number of bytes written on a line before it is replaced. In effect, the higher miss rate of small caches prematurely cleans out cache lines, as partially dirty lines are replaced and then fetched to be written some more. The stride-one numeric applications have the highest number of dirty bytes per victim.

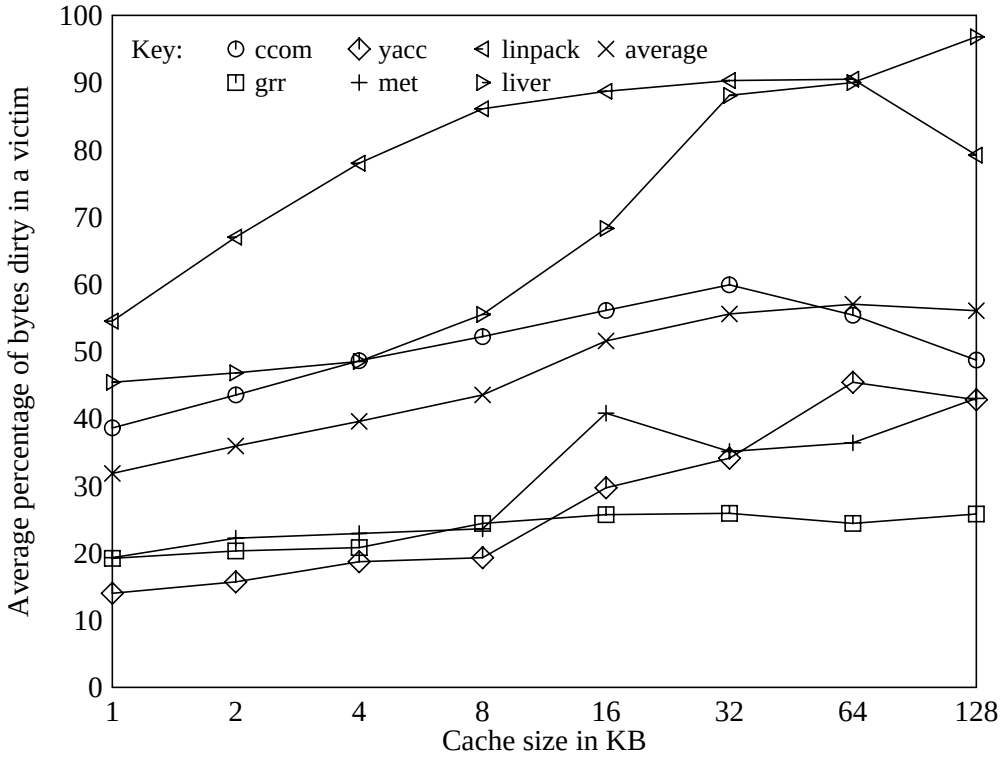

Figure 22: Percent of bytes dirty per victim vs. cache size for 16B lines

Figure 23 gives the percentage of victims dirty vs. line size for 8KB caches. The percentage of victims dirty is about the same or slightly decreasing with increasing line size. This data can tell us about the relative clustering of read and write data. If writes were clustered more than reads, the percentage of dirty victims would decrease with increasing line size. This is because the number of lines required to hold write data would decrease faster than the number of lines required to hold read data. Conversely, if writes were poorly clustered but read data was tightly clustered, the percentage of victims which were dirty should increase with increasing line size. The data in Figure 23 implies that writes are slightly more clustered than reads. This might be a natural consequence of the fact that programs execute about twice as many loads as stores, and that operations usually take more than one input to produce an output.

The percent of bytes dirty in a dirty victim is given in Figure 24. For caches with 4B lines, either all the bytes in a line are clean or they are dirty because the architecture which was simu-

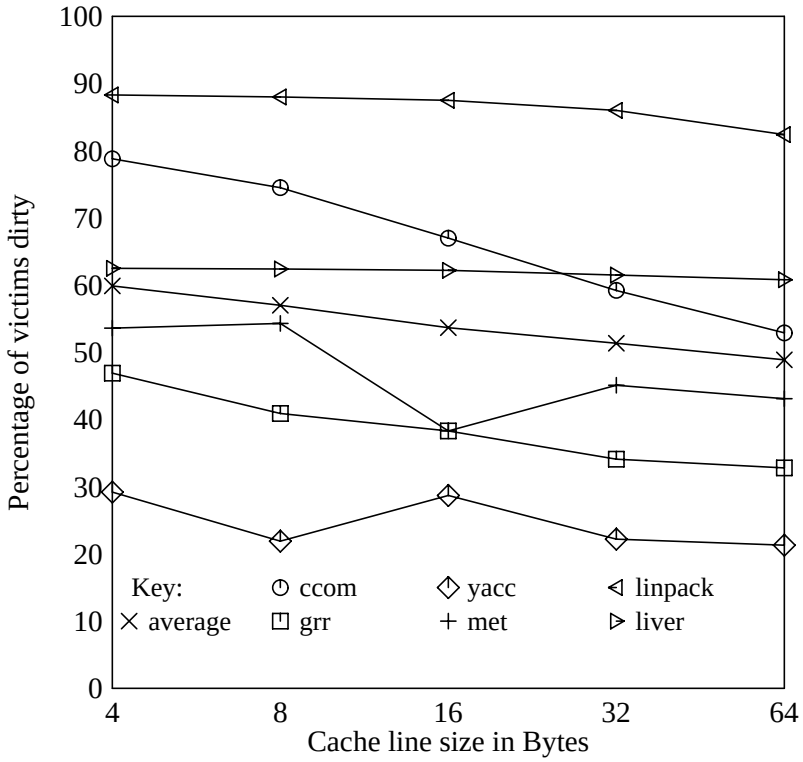

Figure 23: Percent of victims with dirty bytes vs. line size for 8KB caches

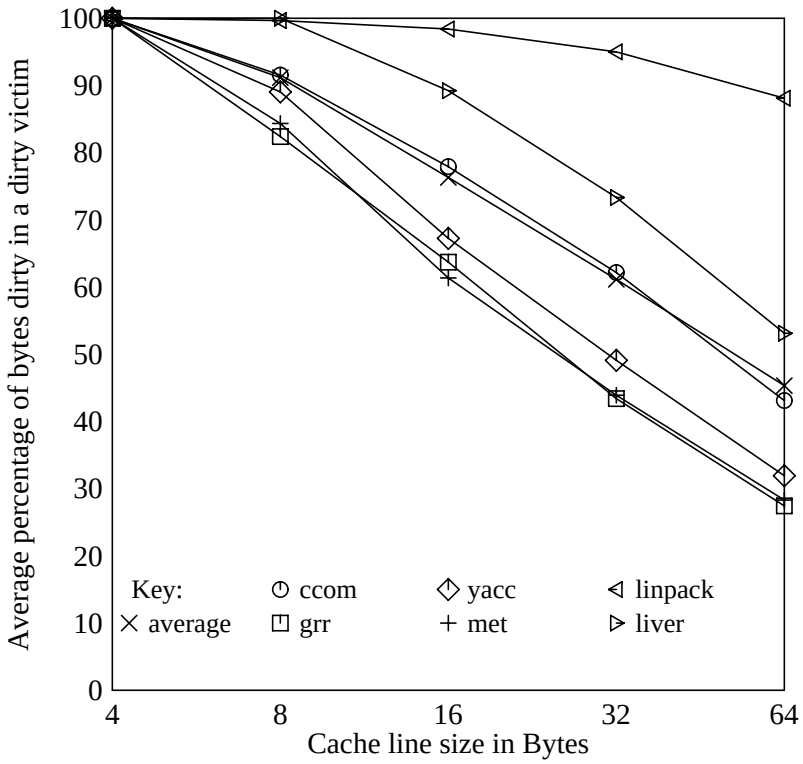

Figure 24: Percent of bytes dirty in a dirty victim vs. line size for 8KB caches

lated does not support byte or halfword writes. For larger line sizes, the percentage of bytes dirty in a dirty line drops off rapidly. This is in keeping with the lower utilization characteristic of longer lines. The numeric benchmarks have the highest percentage of dirty bytes per line, since they have unit stride access. They also have almost 100% bytes dirty in a dirty line for 8B lines, since the vast majority of their writes are stores of double-precision floating-point values.

Figure 25 gives the percentage of bytes on a victim line that are dirty, for a range of line sizes with 8KB caches. Note that this data is averaged over all victims, whether clean or dirty. The average percentage of dirty bytes per victim significantly decreases as the line size increases. This is because as cache lines get larger, a lower percentage of the extra data is useful.

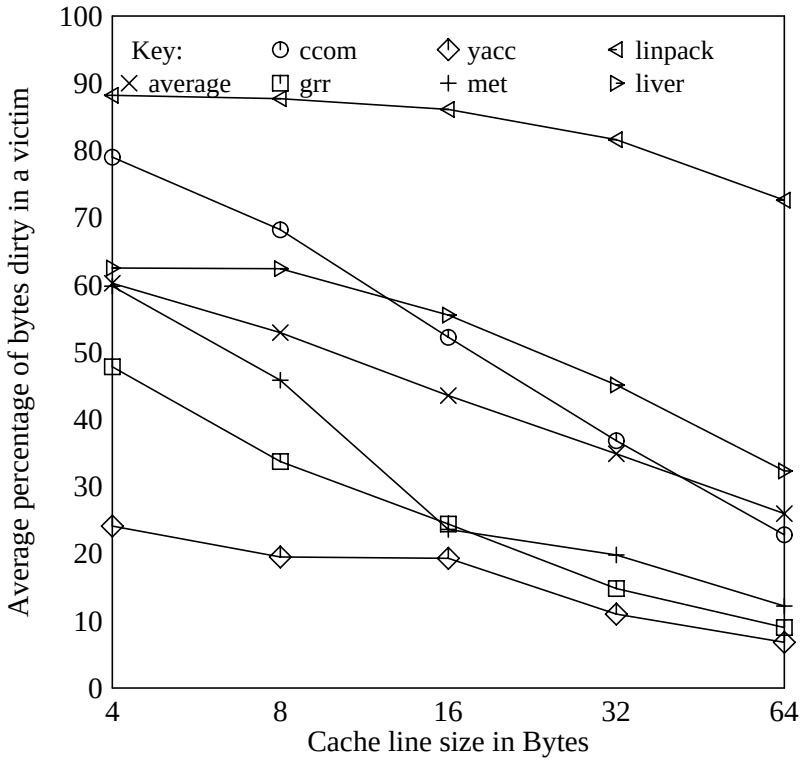

Figure 25: Percent of bytes dirty per victim vs. line size for 8KB caches

Based on data in this section, it appears that an average write bandwidth corresponding to half of the read bandwidth is sufficient, however the actual bandwidth requirements vary widely from benchmark to benchmark. This section did not study the burstiness of dirty victims, which is important when choosing the actual write-back port bandwidth. Since misses are known to be bursty [11], dirty victims are likely to be bursty as well. This would imply that the write back port bandwidth would need to be made wider than the that required by the average bandwidth and/or that buffering to hold more than one dirty victim could be useful.

For cache line sizes of 32B and larger, less than 65% of the bytes on a dirty victim are dirty. This suggests that if write-backs of partial lines are faster than write-backs of whole cache lines, it may be worthwhile to add subblock dirty bits to speedup write-backs.

6. Conclusions

An important issue involving writes that hit in a cache is write-through versus write-back caching. *Write caching*, a technique for reducing the traffic of write-through caches, was studied. It was found that a small fully-associative write cache of five 8B entries could remove 40% of the write traffic on average. This compares favorably to the 58% reduction obtained by a 4KB write-back cache. Since write-through caches have the advantage of only requiring parity for fault tolerance and recovery, while write-back caches require ECC, write-through caches seem preferable for small and moderate sized on-chip caches. Only when cache sizes reach 32KB does the additional traffic reduction provided by write-back caches over write-through caches become significant.

Another area of complexity involving writes is the policy for handling write data on a write miss. Four options exist: either fetch the line before writing (i.e., fetch-on-write), allocate a cache line and write the data while turning off valid bits for the remainder of the line (i.e., write-validate), just write the data into the next lower level of the memory hierarchy leaving the old contexts of the cache line intact (i.e., write-around), or invalidate the cache line and pass the data on to the next lower level in the memory hierarchy (i.e., write-invalidate). Write-validate and write-around always outperform fetch-on-write. In general write-validate outperforms write-around since data just written is more likely to be accessed soon again than data read previously. Write-invalidate always performs worse than write-validate or write-around, but always outperforms fetch-on-write. For systems with caches in the range of 8KB to 128KB with 16B lines, write validate reduced the total number of misses by 30 to 35% on average over the six benchmarks studied as compared to fetch-on-write, write-around reduced the total number of misses by 15 to 25%, and write-invalidate reduced the total number of misses by 10 to 20%. Unlike cache line allocation instructions, write-validate is applicable to all write operations. Moreover, it does not require compiler analysis or program directives, works with various line sizes, does not add any instruction execution overhead, and through the use of sub-block valid bits allows a consistent and correct view of memory to be maintained.

The traffic at the back side of various data cache configurations was also studied. The traffic for write-through caches was found to not vary by more than a factor of 2 over a range in cache sizes from 1KB to 128KB and a range in cache line size from 4B to 64B, since the traffic is dominated by store traffic. Overall, the dirty victim traffic out the back of a write-back cache typically accounted for a third of the traffic out the back of the cache. By adopting more advanced write-miss strategies such as write-validate, the traffic at the back of a write-back cache can be reduced to mostly read misses and dirty victims. In particular, reducing write misses is important because they are more likely to cause processor stalls than the write-back of dirty victims.

For write-back caches, on average about 50% of the victim lines were dirty, although this varied widely based on the program. A relatively constant 70 to 80% of the bytes on 16B lines were found to be dirty over the range of cache sizes from 1KB to 128KB. As line sizes were varied, however, the percentage of dirty bytes on a dirty victim varied from 100% with 4B lines down to an average of 40% with 64B lines. This is due to the lower utilization characteristic of longer cache lines. In systems with line sizes larger than 16B, if write-backs can operate more quickly without all bytes being written back, it may be worthwhile to add subblock dirty bits to speedup write-backs.

Acknowledgements

John Ousterhout provided helpful comments several times over the two years in which this tech report was written. Doug Clark, Joel Emer, and Mary Jo Doherty provided helpful comments on a later draft of this manuscript.

References

[1] Agarwal, Anant. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. PhD thesis, Stanford University, 1987.

[2] Clark, Douglas W. Cache Performance in the VAX 11/780. *ACM Transactions on Computer Systems* 1(1):24-37, February, 1983.

[3] Clark, Douglas W., Bannon, Peter J., and Keller, James B. Measuring VAX 8800 Performance with a Histogram Hardware Monitor. In *The 15th Annual Symposium on Computer Architecture*, pages 176-185. IEEE Computer Society Press, June, 1988.

[4] DeLano, Eric, Walker, Will, Yetter, Jeff, and Forsyth, Mark. A High-Speed Superscalar PA-RISC Processor. In *Compcon Spring*, pages 116-121. IEEE Computer Society Press, February, 1992.

[5] Emer, Joel. Private communication.

[6] Fu, John, Keller, James B., and Haduch, Kenneth J. Aspects of the VAX 8800 C Box Design. *Digital Technical Journal* :41-51, February, 1987.

[7] Goodman, James R. Using Cache Memory to Reduce Processor-Memory Traffic . In *The 10th Annual Symposium on Computer Architecture*, pages 124-131. IEEE Computer Society Press, June, 1983.

[8] Hill, Mark D. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, 1987.

[9] 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.

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

[11] Przybylski, S.A. *Cache Design: A Performance-Directed Approach.* Morgan-Kaufmann, San Mateo, CA, 1990.

[12] Radin, George. The 801 Minicomputer. In *(The First) Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39-47. IEEE Computer Society Press, March, 1982.

[13] Smith, Alan J. Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write-Through. *Journal of the ACM* 26(1):6-27, January, 1979.

[14] Smith, Alan J. Cache Memories. *Computing Surveys* :473-530, September, 1982.

[15] Smith, Alan J. Bibliography and Readings on CPU Cache Memories. *Computer Architecture News* 14(1):22-42, January, 1986.

[16] Smith, Alan J. Second Bibliography on Cache Memories. *Computer Architecture News* 19(4):154-182, June, 1991.

[17] Wall, David W. Global Register Allocation at Link-Time. In *SIGPLAN '86 Conference on Compiler Construction*, pages 264-275. June, 1986.

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.

''Writing Fast X Servers for Dumb Color Frame Buffers.''
Joel McCormack.
WRL Research Report 91/1, February 1991.

''A Simulation Based Study of TLB Performance.''
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

''Analysis of Power Supply Networks in VLSI Circuits.''
Don Stark.
WRL Research Report 91/3, April 1991.

''TurboChannel T1 Adapter.''
David Boggs.
WRL Research Report 91/4, April 1991.

''Procedure Merging with Instruction Caches.''
Scott McFarling.
WRL Research Report 91/5, March 1991.

''Don't Fidget with Widgets, Draw!.''
Joel Bartlett.
WRL Research Report 91/6, May 1991.

''Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

''Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.''
G. May Yip.
WRL Research Report 91/8, June 1991.

''Interleaved Fin Thermal Connectors for Multichip Modules.''
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

''Experience with a Software-defined Machine Architecture.''
David W. Wall.
WRL Research Report 91/10, August 1991.

''Network Locality at the Scale of Processes.''
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

''Cache Write Policies and Performance.''
Norman P.Jouppi.
WRL Research Report 91/12, December 1991.

''Packaging a 150 W Bipolar ECL Microprocessor.''
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

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.

''Systems for Late Code Modification.''
David W. Wall.
WRL Technical Note TN-19, June 1991.

''Unreachable Procedures in Object-oriented Programming.''
Amitabh Srivastava.
WRL Technical Note TN-21, November 1991.

''Cache Replacement with Dynamic Exclusion''
Scott McFarling.
WRL Technical Note TN-22, November 1991.

''Boiling Binary Mixtures at Subatmospheric Pressures''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

''A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach''
John S. Fitch.
WRL Technical Note TN-24, January 1992.