Design Issues and Tradeoffs for Write Buffers

Kevin Skadron and Douglas W. Clark
Department of Computer Science
Princeton University
{*skadron, doug*}*@cs.princeton.edu*

Abstract

Processors with write-through caches typically require a write buffer to hide the write latency to the next level of memory hierarchy and to reduce write traffic. A write buffer can cause processor stalls when it is full, when it contends with a cache miss for access to the next level of the hierarchy, and when it contains the freshest copy of data needed by a load. This paper uses instruction-level simulation of SPEC92 benchmarks to investigate how different write buffer depths, retirement policies, and load-hazard policies affect these three types of write-buffer stalls. Deeper buffers with adequate headroom, lazier retirement policies, and the ability to read data directly from the write buffer combine to substantially reduce write-buffer-induced stalls.

1 Introduction

Processor speeds continue to increase much faster than memory speeds, threatening application performance with increasing stall time for both reads and writes. Current processors attempt to bridge the gap with a variety of old and new techniques: multiple levels of caches, non-blocking loads, prefetching, and write buffers are just a few examples. With a few exceptions, published work in this area focuses on improving the performance of read operations. Since poor write behavior can substantially penalize performance and writes manifestly differ from reads, work to improve memory hierarchy performance must include write-specific techniques. We address some performance issues that arise in the design of processor write buffers.

In a system with a write-through first-level cache, a write buffer has two essential functions: it absorbs processor writes (store instructions) at a rate faster than the next-level cache could, thereby preventing processor stalls; and it aggregates writes to the same cache block, thereby reducing traffic to the next-level cache. These design objectives are unfortunately in conflict. The first function is best fulfilled when the buffer is empty, but the second is best fulfilled when it is full of recently-written blocks. Good write buffer designs achieve a balance between these functions.

This paper considers write buffer designs for systems with at least two levels of cache. Many processors place the first-level (L1) cache on-chip to get the fastest possible hit times, so cycle time plays an important role in the L1 design. This means that L1s often use write-through [15]. The DEC Alpha 21064 [9], 21164 [10], and the SUN UltraSPARC-I [13] all use small, on-chip, write-through, L1s. The large second-level (L2) cache usually uses write-back to minimize main memory traffic. Figure 1 shows a typical system with a small, on-chip, write-through L1, a coalescing write buffer, and an off-chip L2.

Real write buffers can cause stalls where an ideal buffer would not: the buffer can overflow, for example, or cause contention for L2. By separately measuring all the ways in which a write buffer causes stalls, we introduce a framework for analyzing write buffer design. We then identify the benefits and tradeoffs that result from varying several write buffer parameters and policies. Our results, which extend some preliminary work [23], show how write buffer performance depends on these design details.

Literature on write buffers typically appears as part of work focusing on other aspects of memory system design, and so little detailed analysis of write buffer issues is available. Anderson *et al.* [1], Chen [5], and Nagle *et al.* [21] find that write buffers contribute significantly to memory stalls, but do not consider alternative buffer designs. Chen and Baer [6] and Chen and Somani [4] discuss the benefits of read-bypassing write buffers. Jouppi [15] considers cache write policies and discusses some write-buffer issues, especially the policy for writing blocks to L2 (or main memory). He also considers a *write cache*, a write buffer organized as a small, fully associative cache with LRU replacement. Instead of merely buffering writes to memory, the write cache waits until

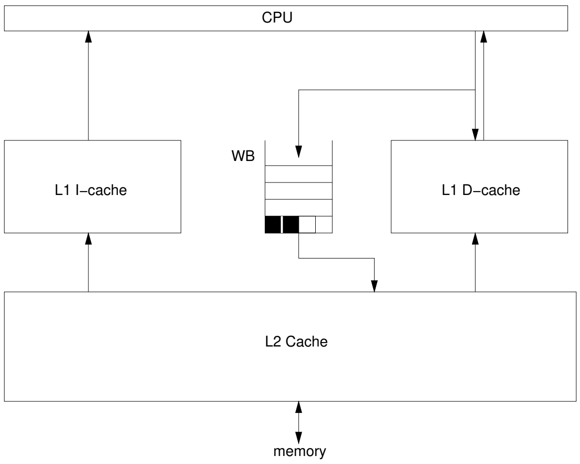

Figure 1: Gross structure of a typical system with two levels of cache. The second level is typically off-chip. We assume a write-through L1 and a coalescing write buffer. The diagram shows a write buffer in which two writes—black boxes—have merged into a single cache line.

Parameter	Value	Hit time	Miss time
Issue	1-way		
Instruction latency	1 cycle, in the absence of memory stalls		
L1 D-cache	8K, direct-mapped, 32B line, write-through, write-around	1	7 read, 1 write
L1 I-cache	perfect	1	na
L2 cache	perfect, write back	6	na (but see Sec. 4)

Table 1: Summary of the machine model. The 1-cycle L1 miss time assumes no write buffer overflow.

Parameter	Description	Baseline value
Depth	The number of entries in the write buffer	4
Width	How many words of data a write buffer entry contains—always 1 for non-coalescing buffers	4 words (32B)
Retirement Order	Which entry is retired when a retirement occurs	FIFO
Retirement Policy	When the front entry (FIFO order) in the write buffer is retired. Based on occupancy in this work	retire-at-2
Load-Hazard Policy	When an L1 load miss hits in the write buffer, dictates how the load obtains its data from the write buffer	flush-full
L2 Priority	Whether reads or writes have priority at L2	Read-bypassing, but underway writes are not preempted

Table 2: Summary of the baseline write buffer model. Experiments focus on depth, retirement policy, and load-hazard policy.

it must evict one of its entries before writing that data to the next level. Bray and Flynn [3] also examine write caches. Gonçalves and Appel [12] study the effects of cache write policy and write buffer configuration on their implementation of the ML language. They find performance of ML programs to be quite sensitive to these design choices. Mowry [20] briefly considers combining a write buffer with a prefetch buffer, but finds little benefit.

A few papers focus more directly on write buffer issues. Smith [24] describes a queuing model for write-through and presents a number of trace-based statistics about write behavior and write buffer depth. Bianchini *et al.* [2], Dahlgren and Stenström [8], Gharachorloo *et al.* [11], Mounes-Toussi and Lilja [19], and Veenstra and Fowler [27] discuss write buffers and write caches in the multiprocessor context. Finally, Chu and Gottipati [7] consider gross write buffer performance. They examine several write buffer configurations for write-through and write-back uniprocessor systems using trace-driven simulation, and find that even a single word of buffering yields a substantial gain in performance.

2 Models, Methods, and Definitions

2.1 Machine Model

We simulate a simple, single-issue machine, with all instructions taking 1 cycle for execution. The memory system adds stall cycles of various kinds. The blocking cache hierarchy has two levels, and inter-cache datapaths are a cache line wide. The 8 K-byte, direct-mapped, write-through L1 data cache has 32-byte lines and uses write-around on write misses. We assume the L1 instruction cache and the write-back, unified L2 never miss (but see Section 4).

Hits in L1 take a single cycle. Write misses, which as mentioned do not allocate in L1, also take a single cycle unless the write buffer is full. L2 has a fixed latency of 6 cycles, similar to the Alpha 21164 [10] and SUN UltraSPARC [13]. In the absence of contention, then, L1 load misses take a total of 7 cycles $(1+6)$ and writing a write-buffer block to L2 takes 6 cycles, regardless of whether the entry being written is full or not. Table 1 summarizes our machine model and the assumed parameters.

This simple model lets us isolate and reason about write buffer effects without the complications of a detailed, low-level model. Trends are intuitive and tradeoffs are clear. Assuming perfect instruction and second-level caches avoids artifacts that specific sizes and organizations might introduce. The price for these simplifications, of course, is that we cannot make absolute quantitative claims. Section 4 explores the model design-space beyond this simple baseline configuration.

2.2 Write Buffer Model

We model a coalescing write buffer of multiple entries, where each entry holds one or more address-aligned words—typically one cache block. Each entry needs an address tag. The write buffer compares these tags in parallel with the address of the incoming store instruction; on a hit, the store merges into the matching entry, and turns on the appropriate word valid bit. On a miss, the store allocates a new entry. If the store finds no entries available, it blocks until the write buffer *retires* an entry to L2. Each entry needs valid bits at the granularity of the smallest writable datum. The DEC Alphas we consider write only 4- or 8-byte words, but new Alphas and most other machines can write quantities as small as a byte.

The write buffer tries to ensure an adequate supply of free entries by retiring entries according to some policy. To retire an entry, the write buffer arbitrates for access to L2, writes those words which are valid, and then marks the entry free. Stores cannot normally merge into an entry that is being retired. Stores can, however, update other buffer entries while a retirement takes place.

Retirement order—typically FIFO—determines which entry gets retired. *Retirement policy* determines when to retire that entry. The Alpha 21064 and 21164 retire the oldest entry if 2 or more entries are valid. A lone entry in the write buffer may remain until it becomes too old—this occurs after 256 cycles in the 21064 [9] and after 64 cycles in the 21164 [10]. Waiting until 2 or more entries are valid before retiring means that sequential writes can achieve maximal coalescing: the most recently allocated entry cannot be retired until a new entry is allocated. We call the entry

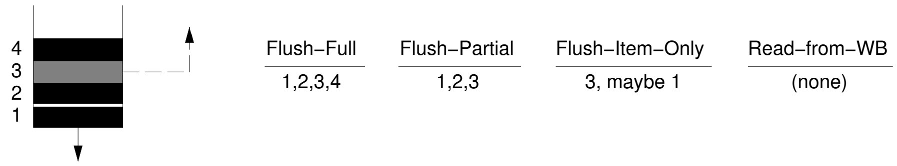

Figure 2: A 4-deep write buffer, in which an L1 load miss has hit the third (gray) entry. The figure shows which entries are flushed by each policy we consider.

Name	Description	How Measured
Buffer-full stall	The write buffer is full and the store cannot merge	Number of cycles the store must wait for a free entry
L2-read-access stall	The write buffer occupies L2	Number of cycles the load must wait to access L2
Load-hazard stall	The cache line needed by an L1 load miss is active in the write buffer	Number of cycles spent handling the load hazard before the load miss can be serviced

Table 3: Summary of write-buffer-induced stalls

that triggers retirement—the second entry in this case—the *high-water mark*, and name the retirement policy according to the high water mark, so the Alphas use *retire-at-2*. Given a sufficiently deep buffer, retirement can be *lazier* (or less *eager*): a 12-deep or 16-deep buffer, for example, could implement retire-at-4, retire-at-8, etc. Lazier retirement keeps entries in the write buffer longer to allow more opportunities for coalescing.

Retirement policy need not be based on occupancy; Jouppi [15] considers a buffer that retires entries at a fixed rate. He finds that in order to avoid overflow, an 8-deep, cache-line-wide buffer must retire entries at a rate too rapid to achieve much coalescing. Occupancy-based policies are not much more complicated than fixed-rate policies, and should always perform better: the buffer can retain a suitable number of entries for coalescing purposes, but can retire entries at the maximum possible rate when occupancy rises above the high-water mark, provided the high-water mark is not too high. Our experiments retire entries based only on occupancy.

The maximum rate at which the write buffer can retire entries depends on the write buffer's priority in arbitrating for L2, and on the write bandwidth at L2. Chen and Baer [6], Chen and Somani [4], and Hennessy and Patterson [22] all describe the benefits of *read-bypassing* write buffers: L1 load misses, which are more urgent, may bypass write buffer entries waiting to retire. With read-bypassing, L1 load misses can tie up L2 and delay write buffer retires, making overflow more likely if stores appear among the loads. The UltraSPARC-I [13] uses read-bypassing until the buffer becomes too full, at which point the write buffer gets priority for L2. Like the Alphas [9, 10], we use straight read-bypassing and we assume that write transactions already underway to L2 cannot be interrupted.

Read-bypassing write buffers raise two correctness issues. Reordering of loads and stores—which also occurs with coalescing—may violate multiprocessor consistency requirements, so current architectures include barrier instructions for ensuring needed ordering properties. (Exceptions don't cause a problem with reordering, because a store only writes its data once it cannot raise an exception.) Allowing loads to bypass the write buffer also means that an L1 load miss may need data from the write buffer. In such a case, reading from L2 would yield stale data. We will call this a *load hazard*. A load hazard occurs even if the word needed by the read miss does not reside in the buffer, but some other portion of that cache line is active: because the version of the line contained by L2 holds some stale words, filling L1 must somehow retrieve those active words from write buffer; otherwise, the fill into L1 would obtain stale data.

Flushing the write buffer on every load miss solves the load hazard problem, but at substantial cost. (We use the term "retirement" to describe an autonomous decision by the write buffer to transfer an entry to L2, and "flushing" to describe a transfer forced by some external event, like a load hazard.) Since the write buffer already has comparators, an L1 load miss can check the write buffer, and take action only on a hit. When a load hazard does occur, a variety of options exist. We consider four *load-hazard policies*. *Flush-full* flushes the entire write buffer when the miss hits in the buffer; the Alpha 21064 uses this policy [9]. *Flush-partial*, the 21164's policy [10], saves some work by flushing entries in FIFO order only as far as necessary to purge the hit entry. *Flush-item-only*, which Chu and Gottipati [7] suggest but do not study, saves even more work by flushing only the hit entry. If a different entry is already being retired when the load hazard occurs, we assume this transaction completes first. Finally, *read-from-WB* allows the load miss to read its data directly from the write buffer without altering the buffer's contents, avoiding an L2 access in the process (no L1 fill occurs).

Figure 2 summarizes the policies, showing how many entries each policy retires. Note that under read-from-WB a load miss can find its cache line active in the write buffer, but find the needed word invalid. This situation requires an L1 fill, which for correctness must merge the incoming line from L2 with the active words from the write buffer.

We assume that under read-from-WB the write buffer and L1 data cache are probed simultaneously, and the data is returned from the appropriate location. Our simulations consequently charge the same amount of time for an L1 hit and a write buffer hit (*i.e.*, the correct word is not in L1 but is in the write buffer). When the correct block resides in the write buffer but the needed word does not, our simulations charge an L2 access, but no extra cycles are charged for merging the data from L2 with those words that are valid in the write buffer.

As a baseline model, we choose a 4-deep, cache-line-wide (32B), read-bypassing write buffer that uses retire-at-2 and flush-full. This closely resembles the Alpha 21064's write buffer [9], lacking only that system's policy of periodic retirement of old entries. The 21164 has a similar buffer that is 6 entries deep and uses flush-partial [10]. Table 2 summarizes some write buffer parameters and our baseline model.

2.3 Write-Buffer-Induced Stalls

Three types of stalls can be blamed on the write buffer. *Buffer-full stalls* occur when the processor executes a store, the write buffer is full, and the store cannot merge with any current entry. The store must stall until a buffer entry is available. An *L2-read-access stall* occurs when a load miss in the L1 data cache encounters a delay in reading from L2 because the write buffer is currently writing to L2. The load stalls for two reasons, first until the write buffer transaction completes, and second until the load's own read from L2 completes. We count only the first as an L2-read-access stall, and charge the second to the miss itself. Finally, a *load-hazard stall* occurs when an L1 load miss finds its data in the write buffer; if the data is not read directly out of the write buffer, the load stalls until the write buffer flushes the necessary entries. Here again, we do not charge the subsequent L2 read time as part of the load-hazard stall, since it should be attributed to the miss instead. In our system model, any stall cycles the write buffer causes must fall into one of these three categories. Note that only the first type delays a store instruction; we find that write buffers often delay loads more than stores.

We separately measure cycles spent on buffer-full, L2-read-access, and load-hazard stalls. (Of course we also simulate and count other sources of memory hierarchy stalls, notably the time for L1 cache misses.) Table 3 summarizes the sources of write-buffer-induced stalls and how we attribute cycles to each. Categorizing write-buffer-induced stall cycles directly and clearly shows the impact of write buffer performance and how much each type of stall contributes. This information helps the designer target design changes: if the chief problem is buffer-full stalls, for example, then deeper buffers should help, while if L2-read-access stalls are a problem, then finding some way to achieve more coalescing is worthwhile. Categorizing stalls also makes it easier to see the tradeoffs associated with varying write buffer design parameters—for example, it shows that making the write buffer deeper reduces buffer-full stalls, but increases in other types of stalls consume part of that gain.

This detailed approach yields better information than gross measures like memory stall CPI or indirect metrics like reduction in write traffic. By counting all stalls, we in effect measure the write buffer against a perfect buffer that never overflows and never delays loads. This represents a lower bound, the best performance a write buffer configuration can achieve. Chu and Gottipati [7] also compare their results to an ideal buffer, but instead measure average cycles per memory reference. They also identify the same three types of write-buffer-induced stalls, but do not measure them.

2.4 Simulation Framework and Benchmarks

We simulate a set of SPEC92 benchmarks with an instruction-level simulator based on Digital's ATOM system [25]. ATOM instruments DEC Alpha executables for OSF. It performs binary-to-binary translation, instrumenting an executable with calls into user-supplied analysis routines. The benchmarks are statically compiled and linked using `cc -migrate` with `-O5 -ifo`[1] optimization for C programs, and `f77` with `-fast`[2] for Fortran programs. Some C programs use `-unsigned`, `-ansi_alias`, or `-assume aligned_objects` and *doduc* uses `-automatic` and `-align records`. Table 4 briefly describes each benchmark and shows the inputs our experiments use.

[1] Inter-file-optimization

[2] Sets `-O4` as well as `assume noaccuracy_sensitive`, `-align dcommons`, and `-math_library fast`

Benchmark	Input	Pct. Loads	Pct. Stores
cc1	`stmt.i`	20.2	10.5
compress	`ref`	22.7	8.6
uncompress	`ref`	22.6	8.4
espresso	`tial.in`	19.6	5.1
li	`8-queens`	28.4	16.2
sc	`loada3`	27.2	11.4
cholsky	na	30.5	12.8
doduc	`short`	22.4	6.8
fft	na	21.2	21.0
fpppp	`short`	33.8	12.7
gmtry	na	35.7	12.4
hydro2d	`short`	21.9	8.7
mdljdp2	`short`	14.5	7.6
mdljsp2	`short`	21.1	6.0
su2cor	`short`	24.3	11.0
tomcatv	na	27.5	8.0
wave5	na	20.8	13.9

Table 4: Summary of the SPEC92 benchmarks used. For benchmark descriptions, see SPEC's Web site [26]. Except for *mdljsp2* and *wave5*, the floating-point benchmarks are double-precision. *Cholsky, fft*, and *gmtry* are kernels from the *nasa7* benchmark.

We chose benchmarks which suffer some degree of write-buffer-induced stalls in order to show how varying the write buffer configuration affects performance. Some SPEC92 benchmarks—*ear, ora, alvinn,* and *eqntott*—suffer virtually no write-buffer stalls in the baseline model, and are not included. We do include one such "uninteresting" benchmark, *espresso*, to show that some configurations produce substantially worse performance. *Spice* and *swm256* presented instrumentation difficulties, and are also omitted. Operating system activity is omitted as well. Of course, we can make no claim for any special representativeness of these benchmarks; they are simply SPEC92 programs.

3 Results and Analysis

This section reports and discusses the performance of different write buffer configurations. We focus on buffer depth, retirement policy, and load-hazard policy. Our general method, once we have presented performance data for the baseline write buffer, is to vary one parameter at a time and examine the results. We do not try to find the "best" structure for our particular machine model; rather we explore the relationships among the parameters and how they affect the three types of stalls.

3.1 Baseline Performance

Figure 3 shows the benchmarks' performance with the baseline write buffer. The figure shows four bars for each benchmark; the first three bars (gray) show the percentage of execution time spent on L2 read-access, buffer-full, and load-hazard stalls respectively, and the fourth bar (black) shows their sum, the total time spent on write-buffer-induced stalls.

Some benchmarks, like *espresso*, stall little, but others—especially the floating-point programs—experience substantial stalls. Nine of the benchmarks, including the three NASA kernels, spend 5% or more of their time on write-buffer-induced stalls. The

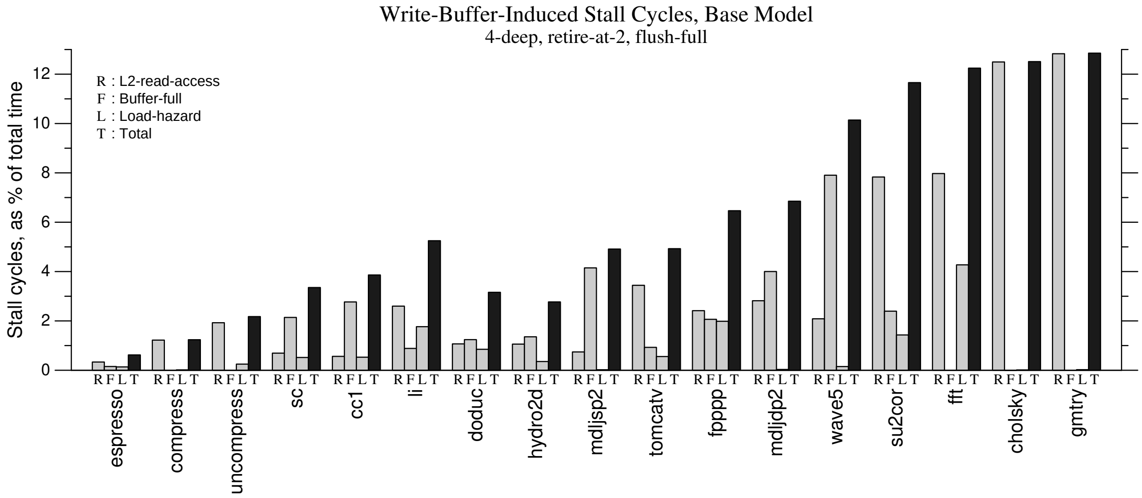

Figure 3: Performance of the baseline write buffer configuration, a 4-deep, cache-line-wide buffer using retire-at-2 and flush-full. The benchmarks are presented in three groups: SPECint92, SPECfp92, and NASA kernels. Within these groups, benchmarks are shown in order of stall behavior.

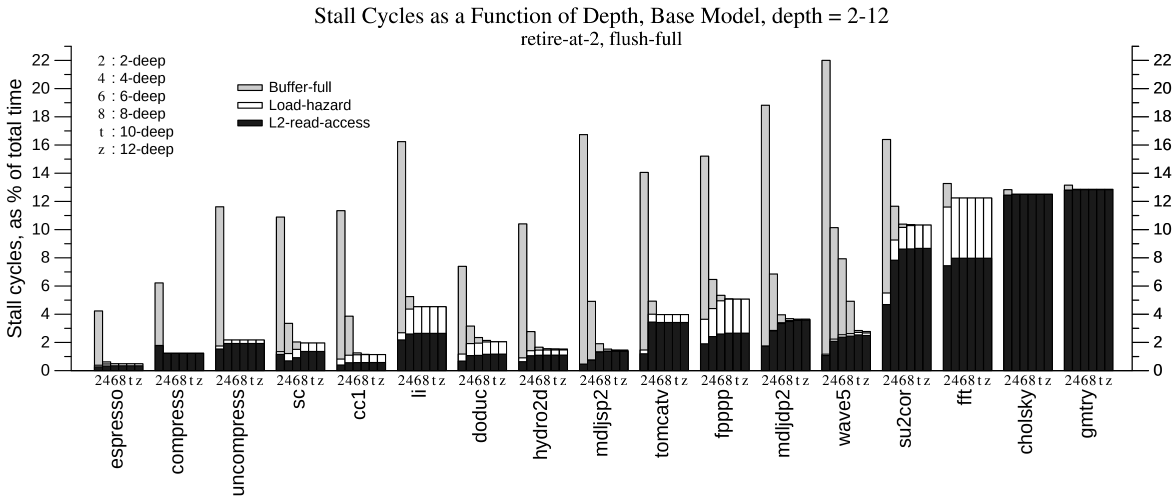

Figure 4: Performance with different buffer depths.

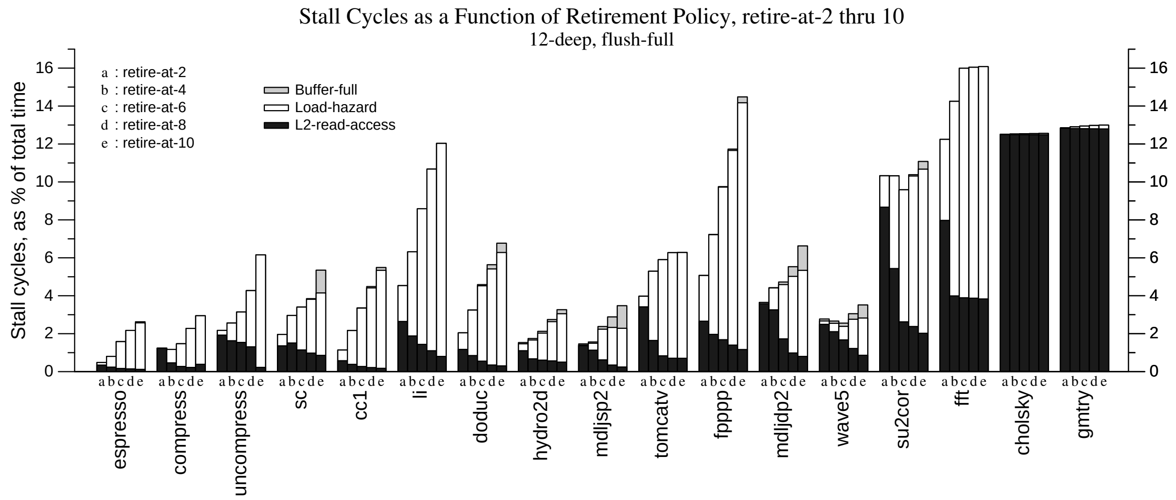

Figure 5: Performance of a 12-deep, flush-full buffer with different retirement policies.

Benchmark	L1 hit rate	WB hit rate
compress	82.52%	38.81%
uncompress	92.10%	21.22%
espresso	94.73%	45.65%
cc1	93.33%	47.46%
li	91.96%	41.40%
sc	91.00%	61.73%
cholsky	48.77%	32.29%
doduc	88.89%	46.65%
fft	57.14%	50.93%
fpppp	89.88%	35.13%
gmtry	43.23%	9.76%
hydro2d	84.29%	44.68%
mdljdp2	85.11%	7.79%
mdljsp2	96.84%	7.41%
su2cor	45.82%	23.56%
tomcatv	63.93%	30.05%
wave5	89.44%	39.32%

Table 5: L1 hit rate (loads) and write buffer hit rate (stores) in baseline model.

incidence of buffer-full and L2-read-access stalls depends partly on a program's locality. Poor locality in L1 leads to L2-read-access stalls when misses contend with write buffer retirements. *Su2cor* and the NASA kernels especially show this behavior. Poor locality in the write buffer leads to buffer-full stalls when non-adjacent stores try to allocate too many buffer entries. *Mdljsp2* and *mdljdp2* are good examples. Table 5 shows the hit rates for the L1 cache (loads only) and write buffer (stores only).

In some cases, obvious transformations can substantially improve cache performance. For example, *gmtry* and *cholsky* both have low L1 and write buffer hit rates. They traverse their arrays in column-major instead of row-major order, the "wrong" order for Fortran, which stores successive column elements sequentially. A loop interchange for *gmtry* and an array transposition for *cholsky* solve the problem—see Table 6. In fact, the new versions suffer almost no write-buffer-induced stalls under the baseline model.

In [16], Lebeck and Wood describe some general techniques for improving cache behavior, such as loop interchange and loop fusion. In addition to the above improvements to *gmtry* and *cholsky*, they obtain good speedups for several other SPEC92 benchmarks, notably *tomcatv*. But they also point out that tuning cache performance is difficult. Most programs are not as easily analyzed as the NASA kernels—finding opportunities for improvement often requires cache profiling [16, 17], and even then conflict misses can make cache behavior vary from input to input. We therefore focus on simple changes to the write buffer itself, and not on compiler techniques or application-specific modifications. Results are for the SPEC benchmarks as shipped, without cache optimizations.

3.2 Buffer Depth

Figure 4 shows how write buffer performance improves as the baseline buffer's depth varies from 2 entries to 12 entries, with all else held constant. Note that because this and all subsequent graphs compare different configurations, each configuration is represented by a single bar. Within a bar, the black segment shows L2 read-access stall cycles as a percentage of execution time, the white shows load-hazard stall cycles, and the grey shows buffer-full stall cycles.

Benchmark	Original kernels		After transformations	
	L1 cache hit rate	WB hit rate	L1 cache hit rate	WB hit rate
gmtry	43.2%	9.8%	88.5%	72.2%
cholsky	48.8%	32.3%	82.1%	73.5%

Table 6: L1 cache and write buffer performance of two NASA kernels, before and after transformations to achieve column-major array traversal in inner loops [16].

The deeper the buffer, the more room for bursts of stores. For most benchmarks, buffer-full stalls cycles fall below 0.2% by the time a depth of 8 is reached. *Wave5* requires 10 entries to reach that level. The NASA kernels have almost no buffer-full stalls, except in a 2-entry buffer, so extra depth has little effect for them. Size-2 buffers perform poorly because under retire-at-2, retirement does not begin until the buffer is full. An actual buffer this small would use more eager retirement.

Deepening the buffer only causes a slight rise in L2 read-access and load-hazard stalls. L2 read-access stalls rise because buffer-full stalls happen less frequently, so more entries retire during non-stalled execution; this raises contention for L2. Load-hazard stalls rise because the average occupancy of the buffer is higher; this raises the probability of a load hazard. These effects are small in comparison to the gains from making the buffer deeper.

3.3 Retirement Policy

Deepening the buffer does more than reduce buffer-full stalls; it also facilitates lazier retirement. Lazier retirement keeps entries in the buffer longer, providing more opportunities for coalescing. If stores were strictly sequential, retire-at-2 would be sufficient: stores would only hit in the most recently created entry.

To evaluate the benefits of different retirement policies, we consider a 12-deep buffer with five different retirement policies: retire-at-2 through retire-at-10. Figure 5 shows the results. For almost every benchmark (not including the idiosyncratic *cholsky* and *gmtry*), lazier retirement has two effects, both due to the longer presence of more blocks in the buffer. First, L2-read-access stalls decrease, because more coalescing makes retirements less frequent (stores are not strictly sequential). Second, load-hazard stalls increase, both because more loads hit in the buffer, and because the cost of each hit—a flush of all occupied blocks—rises. In the buffer configurations of Figure 5, the full flush causes the second effect to overwhelm the first. The next subsection shows that alternatives to flush-full reduce load-hazard stall time.

Figure 5 shows a third effect. Despite using a 12-deep buffer, buffer-full stalls reappear for some benchmarks when using retire-at-8 or retire-at-10—even though Figure 4 suggested that a 12-deep buffer is big enough for these benchmarks to entirely avoid buffer-full stalls. The problem is that, for many benchmarks, retire-at-10 and to some extent retire-at-8 do not leave adequate *headroom*—once the high-water mark is reached, not enough entries remain available. These data suggest that no matter what the retirement policy, the buffer needs at least 4 to 6 entries above the high-water mark to avoid buffer-full stalls for these benchmarks.

3.4 Load-Hazard Policy

Figure 6 shows the effects of different load-hazard policies for

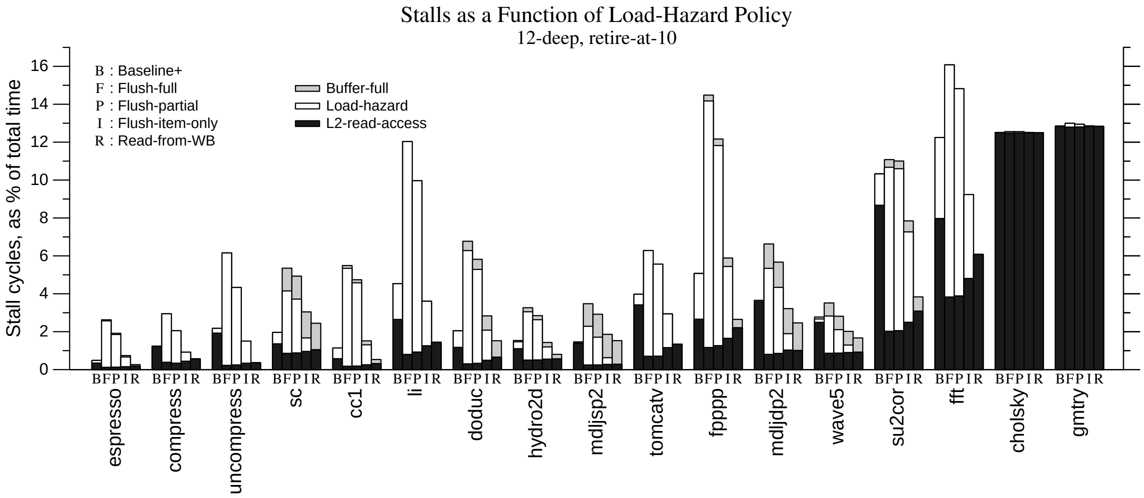

Figure 6: Performance of a 12-deep, retire-at-10 buffer with different load-hazard policies. The first bar for each benchmark, "Baseline+", shows performance for a 12-deep, retire-at-2 buffer with flush-full for comparison.

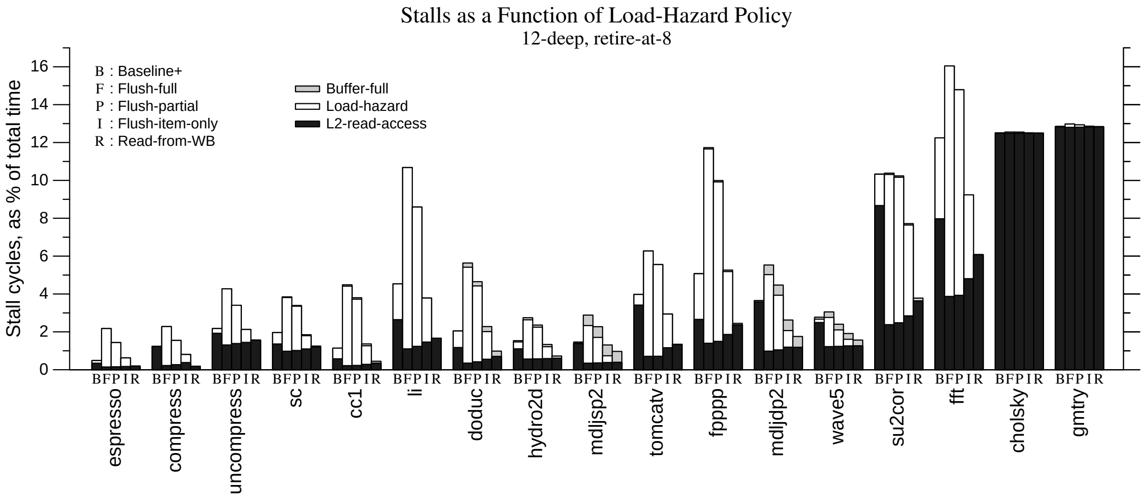

Figure 7: Performance of a 12-deep, retire-at-8 buffer with different load-hazard policies. "Baseline+" is the same as in the previous figure.

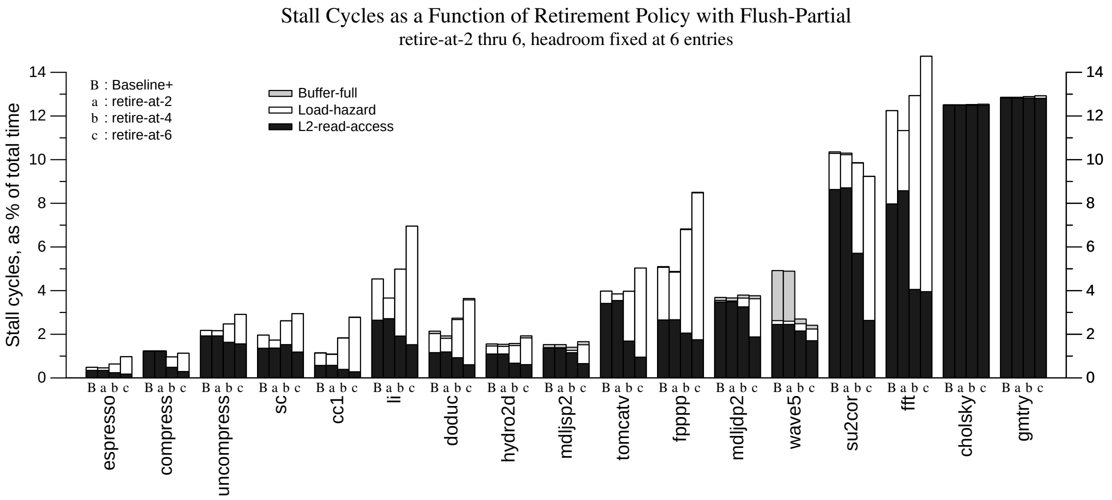

Figure 8: Finding the best configuration for flush-partial.

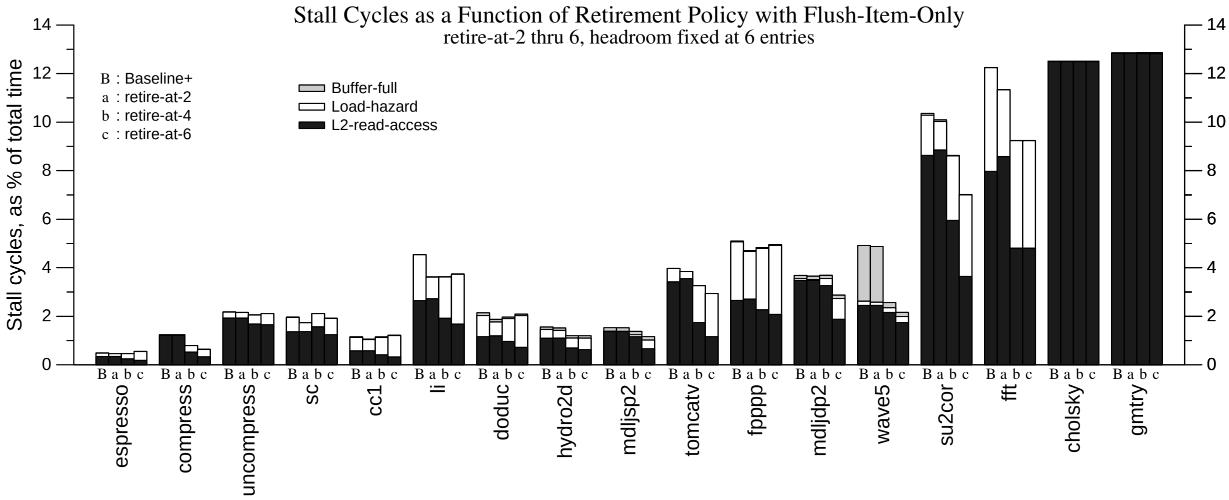

Figure 9: Finding the best configuration for flush-item-only.

a 12-deep buffer with retire-at-10. We choose this low-headroom configuration to expose the interaction between load-hazard policy and headroom; we just saw that a retire-at-10 write buffer does not have enough headroom and so incurs some buffer-full stalls. The first bar for each benchmark, "baseline+", appears for comparison. It shows performance for a 12-deep, retire-at-2 buffer with flush-full—just a baseline buffer with more entries.

The policies introduced in Section 2.2 determine what happens when an L1 load miss hits an entry in the write buffer: in flush-full all active entries are flushed; in flush-partial the hit entry and any older entries are flushed; in flush-item-only the hit entry alone is flushed; and in read-from-WB the desired data is delivered to the processor directly, with no flushing whatever. We say that this progression of policies becomes more *precise* as fewer blocks are flushed.

Increasing precision affects stalls in four ways: first, the duration of load-hazard stalls decreases due to the smaller number of flushed blocks; second, the number of load-hazard stalls increases, because those blocks not flushed can trigger subsequent load hazards; third, L2-read-access stalls increase, since those blocks not flushed are ultimately written on retirement and contention for L2 rises; and fourth, buffer-full stalls may rise, as more blocks increase the buffer's average occupancy. In the configuration of Figure 6, the first effect strongly dominates the others for all but the two pathological NASA kernels. More precise load-hazard policies lead to many fewer load-hazard stall cycles.

More precise load-hazard policies further increase headroom pressure. In the previous subsection, we saw that lazier, low-headroom policies like retire-at-10 in a 12-deep, flush-full buffer cause a moderate rise in buffer-full stalls. Figure 6 shows that buffer-full stalls rise further with more precise flushing. The higher average occupancy creates a greater need for adequate headroom.

Figure 7 repeats the experiments of Figure 6, but with a somewhat more eager retirement policy, retire-at-8. The resulting headroom of 4 blocks nearly eliminates buffer-full stalls. L2-read-access stalls are the same or greater in Figure 7 than in Figure 6, due to less coalescing and hence more frequent retirements. Load-hazard stalls are the same or fewer, due to the lower flush cost of load hazards. Once again, more precise flushing increases headroom pressure; for a few benchmarks, buffer-full stalls rise slightly as the load-hazard policy moves from flush-full to read-from-WB. Overall, the buffer structures of Figure 7 have somewhat better performance than those of Figure 6. *Uncompress, cholsky*, and *gmtry* are exceptions.

These results suggest three conclusions. First, when using a load-hazard policy other than read-from-WB, lazier retirement is worse than eager retirement: we just saw that retire-at-8 outperforms retire-at-10, and that retire-at-2 (even with flush-full, *i.e.* baseline+) outperforms both. Second, the load-hazard policies of intermediate precision—flush-partial and flush-item-only—are not especially helpful. They don't reduce load-hazard stalls enough with lazier retirement, and they don't help much with eager retirement because there are few load-hazard stalls to reduce. (But there are a few exceptions, namely *li, fpppp, su2cor*, and *fft*.) Third, when using read-from-WB, lazier retirement does help—significantly. A 12-deep buffer with retire-at-8 and read-from-WB is the best configuration so far.

3.5 Putting It All Together

The preceding results suggest that lazier retirement only helps in conjunction with read-from-WB. This subsection gives a more detailed picture of the tradeoffs between lazier retirement and more precise load-hazard policies.

To show that retire-at-2 is indeed the best retirement policy for flush-partial and flush-item-only requires experiments like the one that produced Figure 5. Those experiments varied retirement policy, and showed that lazier retirement clearly hurts under flush-full. Figures 8 and 9 show the consequences of different retirement policies under flush-partial and flush-item-only. These experiments are different in one important respect: we saw earlier that write-buffer headroom matters more than simple depth, so in these figures retirement policy varies while headroom remains fixed at 6 entries—depth therefore varies, too.

Figure 8 shows that flush-partial behaves similarly to flush-full. With any but the most eager retirement policy, load-hazard stalls outweigh any gains from lazier retirement and greater depth. Nevertheless, in a retire-at-2 (*i.e.*, 8-deep) configuration, flush-partial performs at least as well as flush-full because it does flush fewer blocks, It provides better performance for several benchmarks, especially *li* and *fft*.

Figure 9 shows mixed results for flush-item-only. In a retire-at-2 configuration, flush-item-only performs about as well as flush-partial. With lazier retirement, performance improves slightly compared to baseline+ for some programs, and significantly for a few. For others, however, performance changes barely at all.

These results show that only the most eager retirement policy, retire-at-2, is viable for flush-full and flush-partial. For flush-item-only, lazier retirement does help some programs.

With read-from-WB, lazier retirement always helps. Read-from-WB simply eliminates load-hazard stalls, so any read-from-WB configuration with adequate headroom must outperform a similar configuration that needs to flush blocks. (Never flushing means read-from-WB needs to retire more blocks, but flushing a block is at least as expensive as retiring it.)

For these benchmarks we conclude that a write buffer should use a deep, read-from-WB buffer with at least 4 to 6 entries of headroom. If hardware exigencies make read-from-WB too difficult, flush-item-only may be an option, but a simple 6-deep or 8-deep, flush-full buffer using retire-at-2 may be better.

4 Cache effects

We next look at the sensitivity of write-buffer performance to some size and latency parameters of the memory hierarchy components.

4.1 L1 Caches

Several current machines—for example, the Alpha 21164 [10] and the Pentium Pro [14]—have 8K, on-chip L1s like our model, but future machines will probably use larger caches. Already, the SUN UltraSPARC [13] has a 16K cache, and the MIPS R10000 [18] a 32K cache. What should happen to the write buffer as L1 size increases? We can identify three effects:

1. Fewer loads miss in L1, so fewer load misses hit in the write buffer and fewer blocks are flushed to L2. This should mean that buffer-full stalls increase, due to increased occupancy of the buffer, but that load-hazard stalls decrease.
2. Fewer load misses means stores come closer together in time. This increased rate of input to the write buffer should lead to an increase in buffer-full stalls.
3. Fewer load misses means a decrease in the number of L2-read-access stalls.

All three effects are evident in Figure 10, which shows write-buffer performance for our baseline model as L1's size increases from 8K to 32K. The strongest effect is the drop in L2-read-access stalls. The net result of increasing L1's size is a small reduction in total write-buffer stalls for most of the benchmarks and a large reduction for a few.

Individual program idiosyncrasies are also evident. In *fpppp*, load-hazard stalls vanish when the cache increases from 16K to 32K; we may infer that the loads causing the hazards at 16K always hit in a cache of size 32K. In *fft*, L2-read-access stalls increase slightly as the cache grows. Its low hit rate remains almost constant in this range of L1 size, but the L1 misses occur at different times and conflict more with retirements.

4.2 L2 Caches and Main Memory

L2 latency

Our experiments assumed a value of 6 cycles for L2 read and write latency, a reasonable number for contemporary machines.

Benchmark	L1	L2 hit rate		
	hit rate	128 K	512K	1 M
compress	82.52%	92.04%	99.98%	99.98%
uncompress	92.10%	98.67%	99.96%	99.96%
espresso	94.73%	99.96%	100.00%	100.00%
cc1	93.33%	99.31%	99.89%	99.98%
li	91.96%	99.18%	99.98%	99.98%
sc	90.98%	97.87%	99.99%	99.99%
cholsky	48.77%	87.00%	94.93%	98.40%
doduc	88.89%	99.97%	99.85%	99.97%
fft	57.14%	62.45%	99.79%	100.00%
fpppp	89.88%	99.87%	100.00%	100.00%
gmtry	43.21%	88.53%	92.80%	96.09%
hydro2d	84.26%	96.64%	99.77%	99.85%
mdljdp2	85.11%	98.77%	99.99%	99.99%
mdljsp2	96.84%	99.79%	100.00%	100.00%
su2cor	45.80%	90.32%	96.65%	98.62%
tomcatv	63.78%	75.10%	75.60%	91.39%
wave5	89.41%	98.25%	99.04%	99.11%

Table 7: L1 and L2 global miss rates. L1 miss rates shown are for a 1M L2; some have declined slightly from Table 5 due to invalidations required to maintain strict inclusion.

Figure 11 shows that the impact of write-buffer-induced stalls in our baseline model is quite sensitive to this parameter. When this value is only 3 cycles, the write buffer does not significantly impede performance—the speed of L2 is so close to that of L1 that little opportunity for contention or buffer-full stalls arises. But as latency grows from 3 to 6 to 10 cycles, write-buffer stall cycles increase dramatically.

The reasons should be clear. If write-buffer retirements and flushes take a longer time in L2, then all three types of stall should increase: buffer-full stalls because the buffer empties more slowly, load-hazard stalls because flushes take longer, and L2-read-access because load misses must wait longer before they may access L2.

L2 size

We have thus far assumed a perfect L2. If we substitute into our baseline model L2 caches with a 6-cycle latency, realistic current size—128K through 1M—and a main memory latency of 25 cycles, we get Figure 12. Many of the programs show little effect, due to high hit rates in even the smallest L2s—see Table 7. Other programs show the expected rise in write buffer stalls as L2 size declines from the perfect model.

Several benchmarks in Figure 12, however, display a surprising decrease in L2-read-access stalls as L2 size is reduced. Table 7 provides one explanation for this: in these benchmarks, some steps in L2 size cause big changes in hit rate. In a run with many more 25-cycle L2 misses, the percentage contribution of write buffer stalls to execution time can go down: the added L2 miss times simply overwhelm the write buffer's contribution. For example, the big jump in misses in *tomcatv* when the cache goes from 1M to 512K results in the peculiar results for that program in Figure 12. *Fft*, *cholsky*, and *gmtry* display similar, if less striking, behavior.

There is a second explanation for this apparent anomaly. When the L2 miss rate jumps significantly, there are more opportunities for the write buffer to retire entries during these L2 misses (our

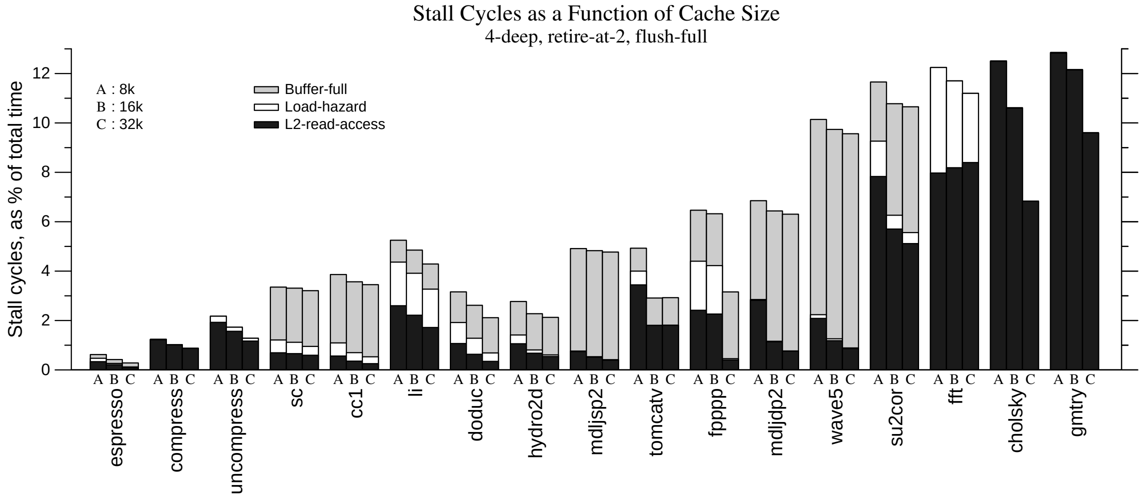

Figure 10: Performance with different L1 sizes.

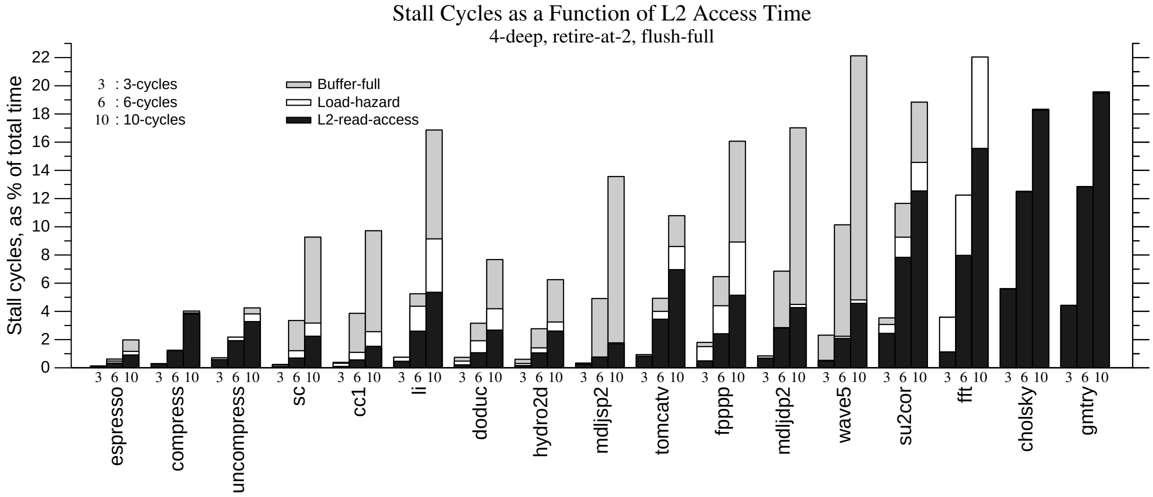

Figure 11: Performance under the baseline write buffer configuration, but with different values of L2 latency.

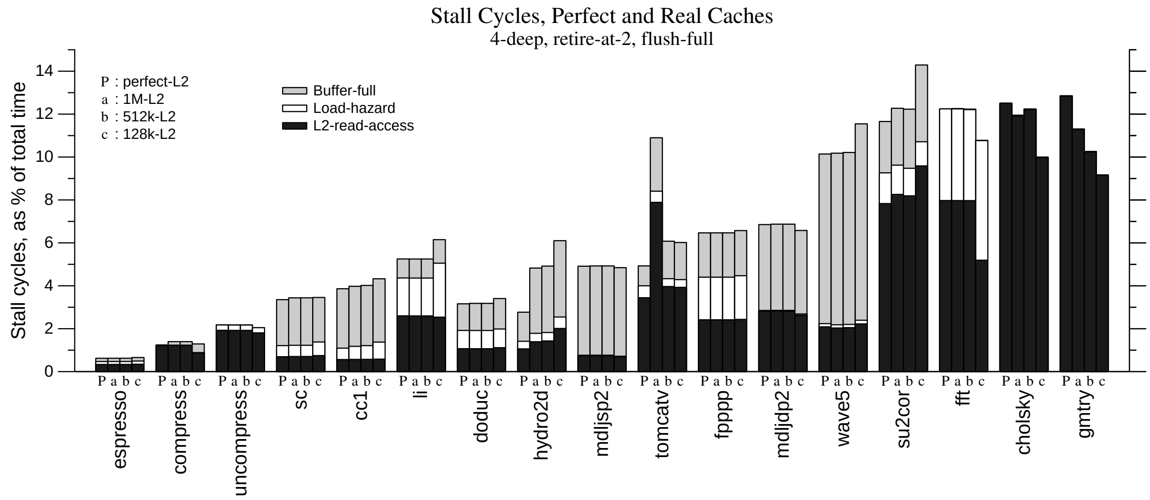

Figure 12: Performance with perfect and real L2 caches of various sizes, and latency 6 cycles.

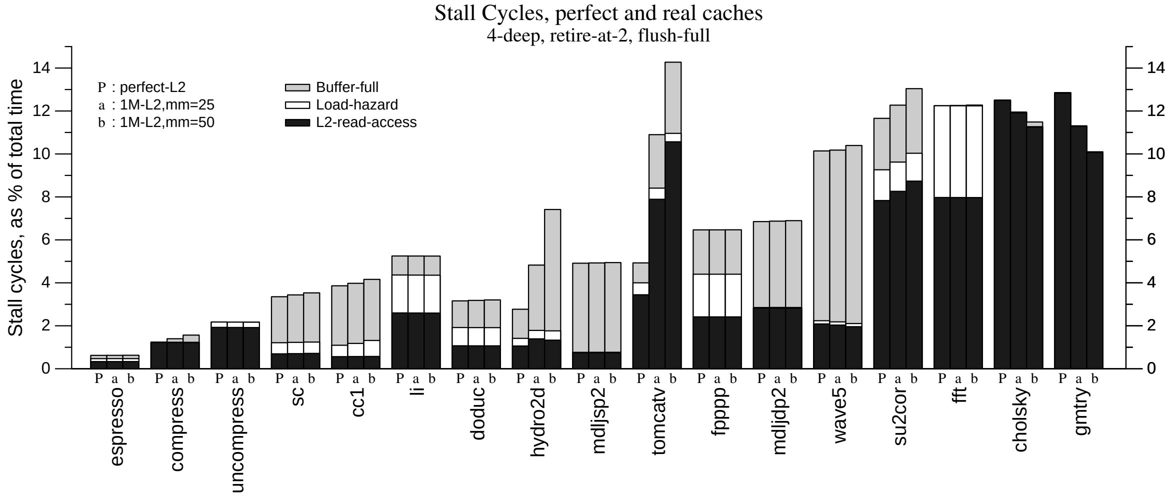

Figure 13: Performance with perfect and real L2 caches with different main-memory latencies.

simulations allow the write buffer to use L2 for retirements then). These retirements cannot conflict with load accesses to L2. Both explanations apply to *tomcatv*: the percentage effect mentioned above exaggerates an *absolute* difference in the number of L2-read-access stall cycles, which is greater for the 1M cache than for the 512K cache.

Main memory latency

Figure 13 shows the differences between a perfect L2, a 1M L2 with memory latency of 25 cycles, and the same 1M cache with memory latency of 50 cycles. Some benchmarks (e.g., *uncompress*, *fpppp*) show little effect, as Table 7 would lead us to expect. In others (e.g., *hydro2d*, *tomcatv*) the latency of main memory has a strong effect on write buffer performance. And in *cholsky* and *gmtry*, the effects discussed above in connection with L2 sizes result in a decrease in the percentage of cycles accounted for by write-buffer stalls.

4.3 Other Effects

Machine organization features that we do not consider in these experiments include the following:

- Substituting a realistic instruction cache for the perfect one used in the simulations has several effects. Stalls due to I-cache misses allow the write buffer to finish retiring an entry if a retirement is underway, and the increase in program runtime reduces the contribution of a given number of write-buffer-induced stalls to total execution time. On the other hand, L2-read-access stalls increase: contention for L2 rises with a real I-cache. An I-cache miss may conflict with a D-cache miss—this cannot be a write-buffer-induced stall with blocking caches—or an I-cache miss may conflict with a write buffer retire. The latter case might be counted as a new category of write-buffer-induced stall: an *L2-I-fetch stall*.
- Our instruction-level simulation does not model stalls due to pipeline data dependencies. Pipeline bubbles spread out stores, so that the write buffer sees a lower store rate and is less likely to overflow.
- We assume datapaths that are a cache line wide, but current machines like the Alphas [9, 10], the UltraSPARC [13], and others have datapaths only half a cache line wide. Narrower datapaths mean that write buffer retirements and flushes take longer, increasing all three types of stalls.
- When caches are non-blocking, L2 read-access and load-hazard stalls can be overlapped with other computation, reducing their impact on processor performance. But the ability to continue executing during cache misses means stores arrive more quickly. Not only does the rate of stores rise, but the possibility of overflow is exacerbated because continued execution increases the rate of loads and load misses. The resulting increase in contention at L2 delays write buffer retirements. This makes write buffer overflows more likely and increases the duration of load-hazards.
- Current machines have varying degrees of superscalarness. The Alpha 21164 [10], for example, can issue 4 instructions per cycle. All else being equal, as issue width increases, store density increases. Write-buffer-induced stalls rise as a result.
- A final effect we did not examine is the possible extra cost under read-from-WB of loads that hit the write buffer. We have assumed that these can be just as fast as an L1 cache hit. If hardware constraints prevent this, the read-from-WB policy will generate short load-hazard stalls.

5 Conclusions

Write-through caches typically require a write buffer to hide the write latency of the next level and reduce write traffic. An empty write buffer is best for the first function, while a full one is best for the second. In this paper we have explored several dimensions of the write-buffer design space in the context of a simple machine model, and have presented simulation results to show how the three types of stalls attributable to write buffers are affected by their design parameters, management policies, and the characteristics of the caches.

We varied write buffer depth, retirement policy, load-hazard policy, and several parameters of the first- and second-level caches. Deeper buffers can essentially eliminate buffer-full stalls, but the headroom—the amount of free space left in the buffer by the retirement policy—matters more than the depth itself. Lazier retirement proves effective at reducing stalls due to L2 contention, but requires a more precise load-hazard policy to avoid losing the gains to sharply increased load-hazard stalls. Among the different load-hazard policies we consider—flush-full, flush-partial, flush-item-only, and read-from-WB—all perform acceptably with eager retirement, but only read-from-WB performs well with lazier retirement. Among the cache parameters we explored, the one with the strongest influence on write-buffer performance was the latency of the second-level cache.

These results, together with our methodology, should serve both as a guide to designers and architects and as a basis for further research in this area.

Acknowledgements

This work was supported in part by NSF grant CCR-9423123 and in part by an NDSEG Graduate Fellowship. We gratefully acknowledge helpful discussions with, and/or comments on drafts of this paper from Pritpal Ahuja, Joel Emer, Kai Li, Margaret Martonosi, Sally McKee, Anne Rogers, Rob Shillner, and J.P. Singh. We also thank the referees for their suggestions.

References

[1] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–20, Apr. 1991.

[2] R. Bianchini, T. LeBlanc, and J. Veenstra. Eliminating useless messages in write-update protocols on scalable multiprocessors. Technical Report TR-539, University of Rochester Department of Computer Science, Nov. 1994.

[3] B. K. Bray and M. J. Flynn. *Specialized Caches to Improve Data Access Performance*. PhD thesis, Stanford Computer Systems Laboratory, May 1993.

[4] C.-H. Chen and A. K. Somani. A unified architectural tradeoff methodology. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 348–57, May 1994.

[5] J. B. Chen. Memory behavior of an X11 window system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 189–200, 1994.

[6] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Oct. 1992.

[7] P. P. Chu and R. Gottipati. Write buffer design for on-chip cache. In *Proceedings of the International Conference on Computer Design*, pages 311–16, 1994.

[8] F. Dahlgren and P. Stenström. Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 26(2):193–210, Apr. 1995.

[9] Digital Semiconductor. *DECchip 21064/21064A Alpha AXP Microprocessors: Hardware Reference Manual*, Jun. 1994.

[10] Digital Semiconductor. *Alpha 21164 Microprocessor: Hardware Reference Manual*, Apr. 1995.

[11] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the Fourth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–57, Apr. 1991.

[12] M. J. R. Gonçalves and A. W. Appel. Cache performance of fast-allocating programs. In *Proceedings of the Seventh International Conference on Functional Programming and Computer Architecture*, pages 293–305, Jun. 1995.

[13] D. Greenley et al. UltraSPARC$^{(TM)}$: The next generation superscalar 64-bit SPARC. In *Proceedings of CompCon '95*, pages 442–50, Mar. 1995.

[14] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, pages 9–15, Feb. 16, 1995.

[15] N. P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, May 1993.

[16] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, pages 15–26, Oct. 1994.

[17] M. Martonosi, A. Gupta, , and T. E. Anderson. Tuning memory performance in sequential and parallel programs. *IEEE Computer*, pages 32–40, Apr. 1995.

[18] MIPS Technologies. *MIPS R10000 Microprocessor User's Manual*, Jun. 1995. Version 1.0.

[19] F. Mounes-Toussi and D. J. Lilja. Write buffer design for cache-coherent shared-memory multiprocessors. In *Proceedings of the International Conference on Computer Design*, pages 506–11, Oct. 1995.

[20] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford Computer Systems Laboratory, Mar. 1994.

[21] D. Nagle, R. Uhlig, T. Mudge, and S. Sechrest. Optimal allocation of on-chip memory for multiple-API operating systems. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 358–69, May 1994.

[22] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*, pages 411–12. Morgan Kaufmann, San Francisco, 2nd edition, 1996.

[23] K. Skadron and D. W. Clark. Measuring the effects of retirement and load-service policies on write buffer performance. In *Proceedings of the 1996 Workshop on Performance Analysis and its Impact on Design*. IBM Austin Research Laboratory, Austin, TX, Mar. 1996.

[24] A. J. Smith. Characterizing the storage process and its effect on the update of main memory by write through. *Journal of the ACM*, 26(1):6–27, Jan. 1979.

[25] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6):196–205, Jun. 1994.

[26] Standard Performance Evaluation Corporation (SPEC). URL: `https://www.specbench.org/`.

[27] J. E. Veenstra and R. J. Fowler. The prospects for on-line hybrid coherency protocols on bus-based multiprocessors. Technical Report TR-490, Univeristy of Rochester Department of Computer Science, Feb. 1994.