Avoiding Store Misses to Fully Modified Cache Blocks

Shiwen Hu

Networking and Computing Systems Group
Freescale Semiconductor, Inc.
7700 W. Parmer Lane, Austin, TX 78729
shiwen.hu@freescale.com

Lizy John

Laboratory for Computer Architecture
The University of Texas at Austin
1 University Station C0803, Austin, TX 78712
ljohn@ece.utexas.edu

Abstract

Memory bandwidth limitation is one of the major impediments to high-performance microprocessors. This paper investigates a class of store misses that can be eliminated to reduce data traffic. Those store misses fetch cache blocks whose original data is never used. If fully overwritten by subsequent stores, those blocks can be installed directly in the cache without accessing lower levels of the memory hierarchy, eliminating the corresponding data traffic. Our results indicate that for a 1MB data cache, 28% of cache misses are avoidable across SPEC CPU INT 2000 benchmarks. We propose a simple hardware mechanism, the Store Fill Buffer (SFB), which directly installs blocks for store misses, and substantially reduces the data traffic. A 16-entry SFB eliminates 16% of overall misses to a 64KB data cache, resulting in 6% speedup. This mechanism enables other bandwidth-hungry techniques to further improve system performance.

1. Introduction

As the speed gap between microprocessor and main memory grows, main memory accesses become a significant bottleneck to processor performance. Memory systems face the problems of long memory access latencies and limited memory bandwidth. Numerous techniques, such as value prediction and speculative execution [7,12], prefetching [2] and multithreading [14], have been proposed to reduce or tolerate long memory access latencies. In return, many of those latency-hiding techniques demand high memory bandwidth, which is already a bottleneck in several systems [3,6,13]. Hence, memory bandwidth limitation becomes one of the major impediments to high-performance microprocessors.

In modern processors, write-allocate caches are usually preferred over non-write-allocate caches [9]. Write-allocate caches fetch and allocate cache blocks upon store misses, while non-write-allocate caches send the data to lower levels of the memory hierarchy without allocating the corresponding blocks. Comparing with non-write-allocate caches, write-allocate caches lead to better performance by exploiting the temporal locality of recently written data [9].

This paper investigates the reduction of memory bandwidth requirement of write-allocate caches by avoiding fetches of fully modified blocks. A store-miss allocated cache block is *fully modified* if 1) the block's original data is never used, and 2) the block is completely overwritten by subsequent stores. Those two properties ensure that the fetches of the original data of fully modified blocks from the lower level of the memory hierarchy can be avoided without affecting program correctness. Accordingly, in this paper, the store misses allocating fully modified blocks are called *avoidable misses*, and the corresponding data traffic is *avoidable data traffic*.

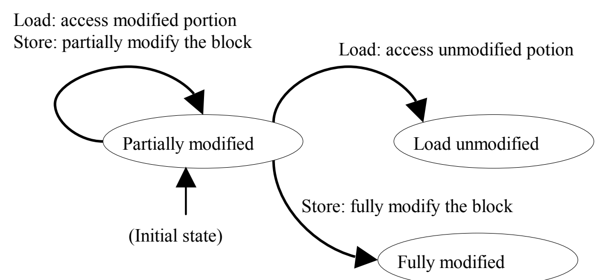

Figure 1. States and transitions of store-miss allocated blocks.

Not all store-miss allocated blocks are fully modified. Those non-fully modified blocks can be further categorized into two types. If a block's original data is read by a load instruction, the block is called *load unmodified*. A non-load-unmodified, non-fully-modified block is *partially modified* since it is evicted from the cache with unmodified portions. Although not used by the processor, the original data of partially modified blocks is still needed to ensure that whole cache blocks, instead of modified block segmentations, are written back to lower levels of the memory hierarchy. Hence, data traffic fetching non-fully-modified blocks cannot be avoided. The states and transitions of

store-miss allocated blocks are illustrated in Figure 1. Initially, a newly allocated block is partially modified.

To reduce the avoidable data traffic, we propose a simple hardware mechanism, the Store Fill Buffer (SFB), which is a small buffer accessed in parallel with the L1 data cache. Data traffic is reduced by directly installing store-miss allocated blocks in the SFB, without accessing lower levels of the memory hierarchy. Compared with previous schemes [8,9,15], this mechanism has advantages such as requiring no compile-time support and incurring minimal hardware overhead. Moreover, by allowing load-miss allocated blocks staying longer in the L1 data cache, a SFB reduces both load and store misses, improving the performance considerably.

This work makes three contributions. 1) We demonstrate that programs usually have abundant avoidable cache misses, regardless of varying cache configurations. For a 1MB data cache, 28% misses that access memory are avoidable. 2) We analyze the various characteristics of store-miss allocated blocks. The results indicate that it is feasible to effectively reduce avoidable misses and data traffic via a low-cost hardware approach. 3) Based on those findings, a hardware mechanism, the Store Fill Buffer, is proposed to reduce data traffic that loads fully modified blocks. With much smaller hardware costs, a SFB reduces more load misses than a write-validate cache, and performs better than a victim cache on overall miss reduction. *A 16*-entry SFB eliminates 16% of overall misses to a 64KB data cache, resulting in 6% speedup across SPEC CPU INT 2000 benchmarks.

The rest of the paper is organized as follows: Section 2 discusses previous efforts in the area, and Section 3 describes the simulation environment and evaluation methodology. The characteristics of avoidable data traffic are presented in Section 4. Section 5 proposes the Store Fill Buffer and evaluates its performance impact. Finally, we conclude in Section 6.

2. Related work

There have been many studies on reducing data traffic. One of such schemes is the write-validate cache [9], in which no data is fetched upon a store miss. Instead, the data is written directly into the cache, and extra valid bits indicate the valid (i.e., modified) portions of the blocks. One of write-validate's deficiencies is the significant implementation overhead, especially when per-byte valid bits are required in architectures such as Alpha [5]. More importantly, a write-validate cache reduces store misses at the expense of increased load misses arising from reading invalid portions of directly installed blocks, negating write-validate's traffic advantage. As a comparison, a SFB reduces both load and store misses, and incurs less hardware overhead to yield better cache performance to a write-validate cache (Section 5.2.2).

Cache installation instructions, such as `dcbz` in PowerPC [8], are proposed to allocate and initialize cache blocks directly [15]. Unfortunately, several limitations prevent broader application of the approach. First, to use the instruction, the compiler must assume a cache block size and ensure that the whole block will be modified. Consequently, executing the program on a machine with wider cache blocks may cause errors. Furthermore, the use of the instruction is limited by the compiler's limited scope since it cannot identify all memory initialization operations.

A hardware mechanism [11] is proposed to identify stores that initialize heap objects, and trigger cache installation instructions to reduce data traffic dynamically. The mechanism's dependence on the system routine `malloc()` limits its application to programs that use the routine exclusively, and can hardly work on other programs, e.g., Java programs. Furthermore, the mechanism cannot identify fully modified blocks arising from program activities other than heap object initialization. In contrast, SFB identifies almost all fully modified blocks with no software assistance, and is effective for programs written in any languages.

Another related scheme is the write cache/buffer [9], which assists write-through caches to coalesce missed stores before written to lower levels of the memory hierarchy. Write-allocate caches, usually employing the write-back policy, rarely use write caches since a write-back cache inherently possesses the capability of write coalescing. Furthermore, a write cache can only reduce the *downward* data traffic, i.e., traffic to lower levels of the memory hierarchy. Since in a write-allocate cache, the data traffic incurred by store misses is *upward*, a write cache is unable to minimize the avoidable data traffic as a SFB.

3. Methodology

This work uses a modified version of the Simplescalar/Alpha version 3.0 toolset [4] to characterize store-miss allocated blocks and evaluate the performance impact of the Store Fill Buffer. Simplescalar/Alpha includes a suite of simulation tools for the Alpha ISA [5], and its timing simulator incorporates a detailed execution-driven out-of-order processor that accurately executes user-level instructions. The baseline machine is configured as an aggressive 8-way out-of-order processor with two levels of caches, as given in Table 1.

Table 1. Configuration of the baseline system.

CPU		Memory Hierarchy	
Instruction window	128-IFQ, 128-RUU, 64-LSQ	**L1 D-cache**	64KB, 64B blocks, 4-way, LRU, 1 cycle hit latency,
Issue/commit width	8 instructions per cycle	**L1 I-cache**	64KB, 64B blocks, 2-way, LRU, 1 cycle hit latency
Functional units	8 intALU, 4 IntMult/Div, 6 FPALU, 2 FPMult/Div	**L2 unified cache**	1MB, 128B blocks, 4-way, LRU, 12 cycles hit latency, 80 cycles miss latency
Branch predictor	2K-entry combined predictor		

Table 2. Characteristics of SPEC CPU INT 2000 benchmarks. (64KB caches, 4-way, 64B blocks)

Benchmark	Input set	L1 d-cache miss rate	Store miss percentage	Benchmark	Input set	L1 d-cache miss rate	Store miss percentage
gzip	log	1.38%	26.57%	**eon**	cook	2.02%	31.52%
vpr	route	2.70%	15.76%	**perlbmk**	diffmail	0.78%	16.36%
gcc	166	6.61%	52.74%	**gap**	ref	4.43%	25.01%
mcf	ref	18.61%	23.02%	**vortex**	two	1.22%	14.70%
crafty	ref	1.31%	12.55%	**bzip2**	program	2.00%	27.81%
parser	ref	2.07%	10.38%	**twolf**	ref	5.48%	17.70%

To perform our evaluation, we collect results from SPEC CPU INT 2000 benchmarks [16]. The benchmarks are compiled with SPEC peak settings, which perform many aggressive optimizations. For each benchmark, the execution of its first billion instructions is fast-forwarded to warm up the simulator, and statistics are collected during the execution of the second billion instructions. Each benchmark's input set, level-one data cache miss rate, and proportion of store misses are summarized in Table 2. On average, 23% of overall misses are store misses for a 64KB L1 data cache. SPEC CPU 2000 floating-point benchmarks are not evaluated in this paper since we could not obtain the Alpha binaries of those benchmarks.

4. Characterizing avoidable misses

In a write-allocate cache, a cache block is allocated due to either a load or a store miss. As discussed in Section 1, a store-miss allocated cache block can be either fully modified, load unmodified, or partially modified. And the store misses allocating fully modified blocks are avoidable since the corresponding data traffic is never used by the program, and can thus be eliminated without affecting program correctness.

In this section, we demonstrate that large amount of store misses are avoidable, regardless of varying cache configurations. We also obtain the various characteristics of store-miss allocated blocks. The results indicate that it is feasible to effectively reduce avoidable misses and data traffic using a low-cost hardware approach.

4.1 Avoidable misses

Figure 2 breaks down store-miss allocated blocks for write-allocate caches ranging from 64KB to 4MB. Load miss rates represent the differences between the top of the accumulated bars and 100% of overall misses. Modern high-performance microprocessors usually have two or more levels of caches, with at least one of them being write-allocate. In this figure, the two smaller sizes (64K and 256K) correspond to L1 data caches, while the two larger sizes (1M and 4M) represent the total capacities of on-chip caches. Hence, the results in Figure 2 indicate the avoidable data traffic between the write-allocate L1 cache and the L2 cache, as well as between the write-allocate L2 cache and the memory.

The store misses allocating fully modified blocks are avoidable. The amount of fully modified blocks is affected by both program characteristics and cache configurations. Since blocks stay longer in a larger cache, many otherwise partially modified blocks become fully modified in a larger cache. Consequently, the proportions of avoidable misses increase as cache size increases. On average, fully modified blocks consist of 14% and 28% of all blocks allocated in a 64K cache and a 1M cache respectively.

Load unmodified blocks represent the extra load misses of a non-write-allocate/write-validate cache over a write-allocate cache. In a write-allocate cache, a store-miss allocated block is load unmodified if a subsequent load accesses its original data. For a non-write-allocate/write-validate cache, such a block is never fetched from the lower level of the memory hierarchy, so the load reference is always missed. Hence, the percentage of load-unmodified blocks of a program implies how well a write-allocate cache out-

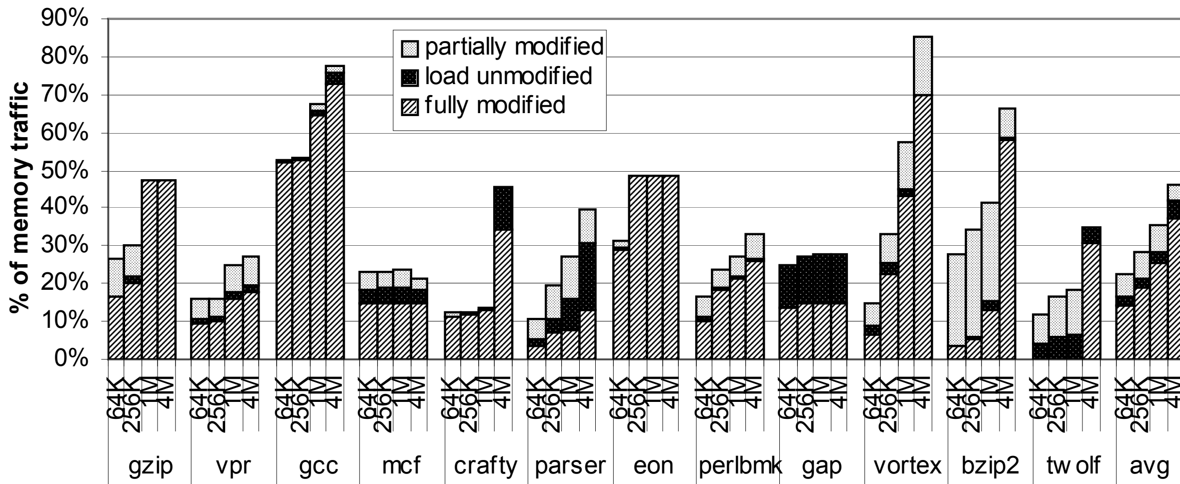

Figure 2. Breakdown of store misses. (Cache parameters: 64KB-4M caches, 4-way, 64B blocks)

performs a non-write-allocate/write-validate cache on the program. Figure 2 shows that many programs have ignorable load unmodified blocks. One distinct program is gap, 11% of whose cache blocks are load unmodified. Hence, a non-write-allocate/write-validate cache will perform badly on the benchmark.

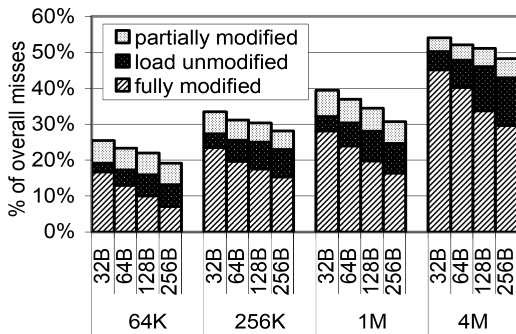

Figure 3. Sensitivity of avoidable store misses to cache sizes and block widths.

4.2 Sensitivity to cache configurations

Figure 3 illustrates the compositions of types of store misses under various cache sizes and block sizes. The results are averaged over all workloads. As the cache block size increases, the proportions of store misses drop, indicating that stores have better spatial locality than loads. Wider cache blocks contain more data, and are intuitively more likely to be partially modified or load unmodified. Consequently, the fraction of fully modified blocks decreases with wider cache blocks. However, even with wide cache blocks, plenty store misses are avoidable. On average 16% of the data traffic is avoidable for a 1MB cache with 256B blocks.

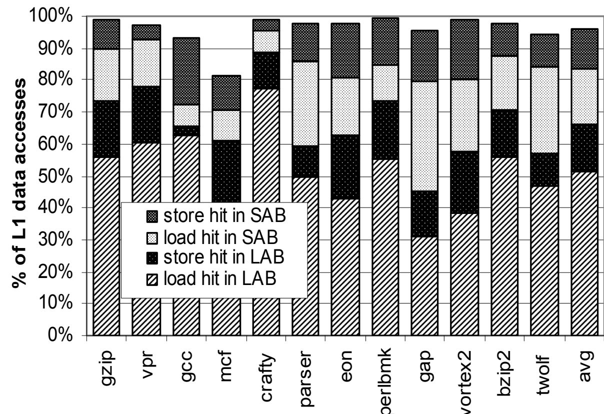

Figure 4. Breakdown of L1 data references. (LAB/SAB – load-miss/store-miss allocated blocks. 64KB cache, 4-way 64B blocks)

4.3 Decomposition of data references

In a write-allocate cache, cache blocks are allocated due to either load or store misses. Loads and stores may access either type of blocks. Figure 4 breaks down the data references by their reference types and the types of blocks accessed by those data references. Hence, each bar consists of four portions, representing the percentages of overall L1D accesses that are loads/stores hitting load-miss/store-miss allocated blocks. Data cache miss rates represent the differences between the top of the accumulated bars and 100% of data references. As shown in Figure 4, accesses to load-miss allocated blocks dominate most

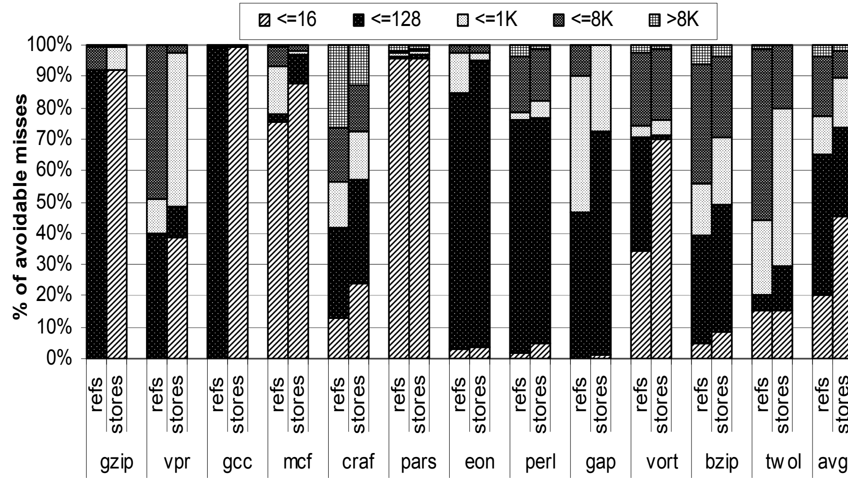

Figure 5. Breakdown of fully modified blocks by their fill intervals. (A block's *fill interval* is the number of data references/stores executed before the block is completely overwritten. Cache parameters: 64KB cache, 4-way, 64B blocks)

benchmarks. On average, 66% and 30% of data references hit load-miss and store-miss allocated blocks respectively, and the other 4% are cache misses to the data cache.

4.4 Fill intervals of fully modified blocks

Figure 5 further categorizes fully modify blocks by the lengths of their fill intervals. A block's *fill interval* is the number of data references/stores executed before the block is completely overwritten. In Figure 5, each benchmark has two bars, representing the breakdown of fill intervals in terms of data references (*refs*) and stores (*stores*) respectively. Since data references contain both loads and stores, a fully modified block's data-reference-based fill interval should be equal to or longer than its store-based fill interval.

The lengths of fill intervals represent the stability of fully modified blocks across different cache configurations. For instance, if a block's lifetime in the cache is equal to or longer than its fill interval, then the block is fully modified. Otherwise, it is partially modified. In this respect, a block with a long fill interval is more likely to be partially modified in case that its lifetime is short. For benchmarks such as `gzip`, `gcc` and `parser`, most of the 64-byte blocks are filled within 16 4-byte stores references, indicating that those blocks are filled up by sequences of successive stores. Heap object initialization is one source of the stores with such good spatial locality [11]. On average, 45% of fully modified blocks are filled with 16 successive stores. Hence, it is feasible to effectively reduce avoidable misses and data traffic using a hardware approach.

5. Eliminating avoidable data traffic

Results in the previous section demonstrate that avoidable data traffic is abundant, and eliminating avoidable data traffic can improve performance by reducing the pressure on store queues and cache hierarchies. In addition, eliminating avoidable data traffic improves memory bandwidth utilization, allowing memory bandwidth hungry techniques, such as prefetching and multithreading, to further boost system performance.

In this section, we propose a simple hardware mechanism, the Store Fill Buffer, to identify fully modified blocks and reduce corresponding data traffic. The Store Fill Buffer assists the L1 data cache to reduce traffic between the L1 and L2 caches. Although not studied in this work, the approach can be applied on the L2 cache to further eliminate data traffic.

5.1 Store Fill Buffer

The Store Fill Buffer (SFB) is a small, fully set associative buffer that is accessed in parallel with the L1 data cache. Having the same block size as the L1 data cache, it uses per-byte valid bits to identify fully modified blocks. The valid overhead can be reduced if the minimum store unit is larger than one byte and all stores are aligned. Considering its small size, the SFB hardly affects the L1 data cache's access latency.

When a store miss occurs, the corresponding block is not fetched from lower levels of memory. Instead, the block is directly installed in a SFB entry. The entry's valid bits are set to indicate the valid/invalid portions. Subsequent stores to the block's

invalid portions update the valid bits. In three cases, a SFB block is evicted to the L1 data cache. 1) The block is fully modified with all valid bits set to one. 2) The block becomes load unmodified when a load reference accesses the block's invalid portion. 3) A new entry is allocated when the SFB is full, and the partially modified block in the LRU SFB entry is evicted to the L1 data cache, leaving the SFB entry to the new block. For fast SFB allocation, a one-entry buffer can temporally hold the evicted partially modified entry until its original data is fetched and allocated in the L1 data cache.

By employing the SFB, data traffic between L1 and L2 caches is reduced since moving fully modified blocks from the SFB to the L1 cache incurs no L2 cache fetches. For load unmodified and partially modified blocks, fetches of their original data are still needed.

SFB affects the system performance in two folds. First, similar to a non-write-allocate/write-validate cache, the load misses to load unmodified blocks incur performance penalty, although Figure 2 indicates that for most programs, such penalty is negligible. On the other hand, with a SFB, load-miss allocated blocks stay longer in the L1 data cache. Since most load references tends to access load-miss allocated blocks frequently (Section 4.3), the longer lifetime of those blocks may improve the L1 data cache locality, and thus system performance, considerably.

The data transfers from the SFB to the L1 data cache are transparent to lower levels of the memory hierarchy, and the L1 data cache still maintains the write-allocate policy. Since both SFB and L1 cache are on chip, such transfers are at full speed. By using a one-entry buffer to temporarily hold the evicted SFB block, the performance penalty of a full SFB can also be minimized.

For multiprocessors, the SFB can be supported by a weak ordering model to maintain cache coherence [1]. Before a block is allocated in the SFB, its update permission should be obtained.

5.2 Evaluation Results

We incorporate the SFB in our simulation environment to evaluate its performance, and compare it with a write-validate cache and a victim cache. As noted in Section 2, a write cache/buffer is rarely used with a write-allocate cache, and cannot reduce the *upward* avoidable data traffic as the SFB. Hence we will not compare the SFB against a write cache.

5.2.1 Cache miss/traffic reduction

For a write-allocate cache, the percentage of cache misses eliminated represents the percentage of L1-L2 data traffic reduced. Figure 6 shows the overall data misses reduced by using the SFBs with 16, 32 or 64 entries (the first three columns), a 32-entry victim cache (the fourth columns) and a 64KB write-validate cache (the fifth columns), respectively. The two bars of each column represent the percentages of load and store misses eliminated respectively, and a negative bar indicates an increase in load misses. Besides the additional 8KB per-byte valid bits, the write-validate cache has the same configuration as the baseline write-allocate cache.

Figure 6 demonstrates that a small SFB is effective on eliminating store misses. For most programs, nearly all fully modified blocks of a 64KB cache (Figure 2) can be recognized by a 16-entry SFB because of the blocks' short fill intervals (Figure 5). However, due to the long fill intervals of `crafty` and `bzip2`, the SFB cannot identify all fully modified blocks of these two programs.

As discussion in Section 5.1, using a SFB may incur extra load misses to unmodified portions of the SFB entries. On the other hand, with a SFB, load-miss allocated blocks stay relatively longer in the L1 data cache, and many conflict misses between load and store-miss allocated blocks are avoided. Both factors being in consideration, Figure 6 shows that most programs' load miss rates are in fact reduced by the SFBs.

On average, a 16-entry SFB improves the cache performance by 16%; 2% of which is the load miss reduction. Larger SFBs are more effective on reducing load misses than store misses.

5.2.2 Comparison with other schemes

Although the SFB is similar to a write-validate cache, write-allocate with the SFB is superior to write-validate for two reasons. First, a SFB incur less hardware overhead (1206B for a 16-entry SFB versus 8KB for a 64KB write-validate cache) and uses on-chip transistors more efficiently than write-validate. Furthermore, a SFB reduces both load and store misses, while a write-validate cache reduces store misses at the expense of increased load misses arising from accessing invalid portions of directly allocated blocks. Since system performance is more sensitive to load misses than to store misses, the increased load misses may negate the traffic advantage of write-validate over write-allocate.

As illustrated in Figure 6, on average, the write-validate cache incurs 16% of overall L1 data cache miss reduction but 2% load miss increase. A 16-entry

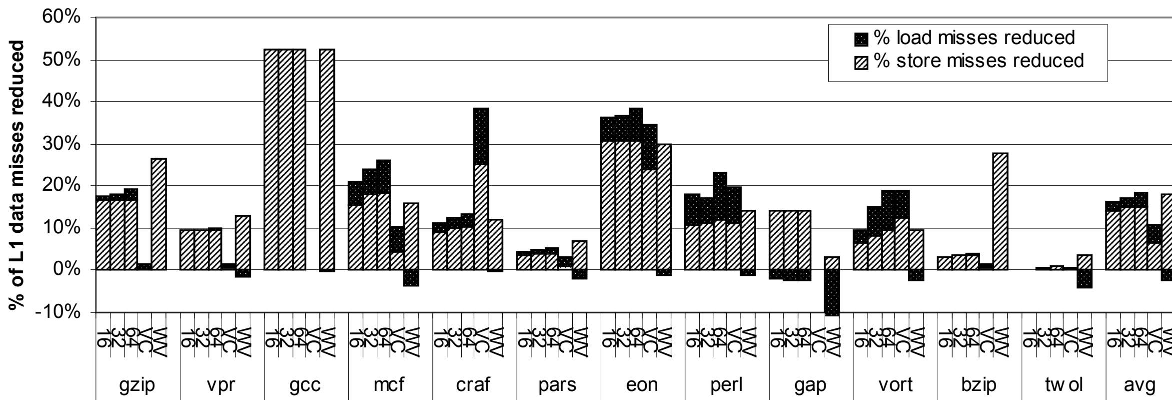

Figure 6. L1 data misses reduced by SFBs, victim cache and write-validate cache. (Each benchmark has five columns: 16/32/64-entry SFBs with 64KB write-allocate caches, a 32-entry victim cache with a 64KB write-allocate cache, and a 64KB write-validate cache)

SFB, with far less hardware cost, achieves similar overall miss reduction and much better load miss reduction.

To justify the increased cache capacity by the SFBs, we compare the SFBs with a 32-entry victim cache [9]. One of the most popular cache assists, a victim cache is a small, fully set associative buffer holding discarded cache blocks. It is checked on cache misses to see if it contains the desired data before going down to the next level of the memory hierarchy. Hence, it is effective in eliminating conflict misses. Figure 6 shows that the 32-entry victim cache reduces the overall miss rates by 11%, less than the 16% achieved by the 16-entry SFB, even though the latter is only half of its size.

5.2.3 Performance impact

Figure 7 compares the performance results, in terms of IPC, of the baseline system (Table 1) and a system combining the baseline configuration with a 16-entry SFB. Differing from what occurs in a conventional write-allocate cache, a store missed in both the data cache and the SFB triggers a direct block allocation in the SFB, which has the same latency as a cache hit unless the SFB is full. In the latter case, the block in the LRU entry of the SFB must be evicted before the new block is installed in the entry. In practice, a one-entry buffer can temporarily store the evicted block to reduce the allocation penalty in a full SFB. A load miss to the invalid portion of a SFB entry incurs the same amount penalty as a L1 load miss.

On average, 6% speedup is achieved by using the SFB. The SFB is especially effective on `gcc` (13% speedup) and `mcf` (27% speedup), which is due to their runtime characteristics such as high miss rates and the abundance of avoidable misses. On the other hand, the SFB has negligible impact on the performance of `perl`, `gap`, and `twolf`.

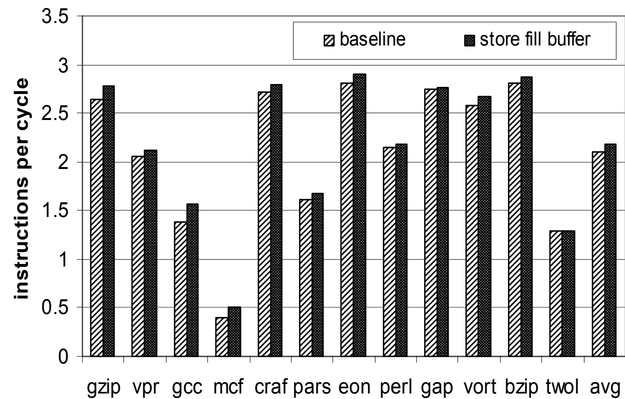

Figure 7. Performance speedup by the Store Fill Buffer. (Baseline configurations with a 16-entry store fill buffer)

6. Conclusions

Memory bandwidth limitation is one of the major impediments to high-performance microprocessors. Hence, reducing memory traffic can improve performance by reducing pressure on store queues and cache hierarchies. It also enables other bandwidth-hungry techniques to further improve performance.

This work investigates the reduction of memory bandwidth requirements of write-allocate caches by avoiding fetches of fully modified blocks. A cache block is fully modified if its original data has not been used until it is fully overwritten by subsequent stores. Hence, without affecting program correctness, those blocks can be directly installed in the cache to reduce

data traffic. The amount of fully modified blocks is affected by both program characteristics and cache configurations. For the SPEC CPU INT 2000 programs, 28% of overall data misses are avoidable for a 1M cache.

We also propose a hardware mechanism, the Store Fill Buffer, to identify fully modified blocks and reduce the data traffic. By delaying fetches for missed stores, the Store Fill Buffer identifies the majority of fully modified blocks even with a size as small as 16 entries. Moreover, the Store Fill Buffer reduces both load and store misses. With significant less hardware cost, the Store Fill Buffer provides comparable cache performance to a write-validate cache. The Store Fill Buffer is also superior to the victim cache in cache performance on overall cache miss reduction. For a 64KB data cache with a 16-entry Store Fill Buffer, on average 16% data misses are eliminated, which results in 6% performance speedup across SPEC CPU INT 2000 benchmarks.

Acknowledgements

The authors would like to thank the anonymous reviewers and Ajay Joshi for their valuable comments. This research was supported in part by NSF grant 0429806, and by IBM, Intel and AMD Corporations.

References

[1]. S. Adve and M. Hill, “Weak Ordering - A New Definition”, in *Proc. ISCA’17*, 1990, pp. 2-14.

[2]. A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng, “Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations”, in *Proc. ICS’ 15*, 2001, pp. 486-500.

[3]. D. Burger, J. Goodman, and A. Kagi, “Memory Bandwidth Limitations of Future Microprocessors”, in *Proc. ISCA’23,* 1996, pp. 78-89.

[4]. D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0”, *Technical Report CS-1342*, University of Wisconsin-Madison, 1997.

[5]. Digital Equipment Corporation, “Alpha 21164 Microprocessor Hardware Reference Manual”, *Maynard Mass*., Apr. 1995.

[6]. C. Ding and K. Kennedy, “Memory Bandwidth Bottleneck and its Amelioration by a Compiler”, in *Proc. IPDPS*, 2000, pp. 181-190.

[7]. F. Gabbay and A. Mendelson, “Speculative Execution Based on Value Prediction”, *Technical Report, Technion*, 1996.

[8]. IBM Microelectronics and Motorola Corporation, “PowerPC Microprocessor Family: The Programming Environments”, *Motorola Inc.*, 1994.

[9]. N. Jouppi, “Cache Write Policies and Performance”, in *ACM SIGARCH Computer Architecture News*, V.21, No.2, May 1993, pp. 191-201.

[10]. N. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, in *Proc. ISCA’90*, 1990, pp 364-373.

[11]. J. Lewis, B. Black, and M. Lipasti, “Avoiding Initialization Misses to the Heap”, in *Proc. ISCA’29*, 2002, pp. 183-194.

[12]. M. Lipasti and J. Shen, “Exceeding the Dataflow Limit via Value Prediction”, in *Proc. MICRO’29,* 1996, pp. 226-237.

[13]. S. Perl and R. Sites, “Studies of Windows NT Performance Using Dynamic Execution Traces”, in *ACM SIGOPS Operating System Reviews*, V.30, No.10, Oct. 1996, pp 169-183.

[14]. D. Tullsen, S. Eggers, and H. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, in *Proc. ISCA’ 22*, 1995, pp. 392–403.

[15]. W. Wulf and S. McKee, “Hitting the Memory Wall: Implications of the Obvious”, in *ACM Computer Architecture News*, V.23, No.1, 1995, pp. 20-24.

[16]. SPEC System Performance Evaluation Committee, http://www.spec.org.