Avoiding Store Misses to Fully Modified Cache Blocks

Abstract

This paper investigates a class of store misses that can be eliminated to reduce memory bandwidth requirements of write-allocate caches. Those avoidable misses allocate cache blocks whose original data is never used until the whole blocks are overwritten by subsequent stores. Hence, those fully modified blocks can be allocated directly in the data cache without accessing lower levels of memory. Our results indicate that for a 1MB data cache, 28% misses to memory are avoidable across SPEC CINT 2000 benchmarks. We also propose a hardware mechanism, the Store Fill Buffer, which can effectively identify avoidable misses and allocate the associate blocks directly to reduce memory traffic substantially. By utilizing a 16-entry Store Fill Buffer, on average 16% overall misses to a 64KB data cache are eliminated, which results in 6% speedup in performance.

1. Introduction

As the speed gap between microprocessor and main memory grows, main memory accesses become a significant bottleneck to processor performance. The inability of memory system is represented by long memory accesses latencies and limited memory bandwidth. Numerous techniques, such as value prediction and speculative execution [7,12], prefetching [2], and multithreading [14], are proposed to reduce or tolerate large memory access latencies. However, many of those latency-hiding techniques demand high memory bandwidth, which is reported to be a bottleneck in several systems [6,13]. Furthermore, the performance of future microprocessors is highly likely to be affected by limited pin bandwidth [3]. Hence, memory bandwidth limitation will be one of the major impediments to future microprocessors.

In modern processors, write-allocate caches are normally preferred over non-write-allocate caches. Write-allocate caches fetches blocks upon store misses, while non-write-allocate caches send the written data to lower levels of memory 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 work investigates the reduction of memory bandwidth requirements of write-allocate caches by avoiding fetches of fully modified blocks. A store allocated cache block is *fully modified* if its original data has not been used until the block is fully overwritten by subsequent stores. Hence, the fetches of fully modified blocks can be avoided without affecting program correctness. Accordingly, the store misses allocating fully modified blocks are called *avoidable misses*, and the associated memory traffic is *avoidable memory traffic*.

Besides fully modified blocks, store allocated blocks can be categorized into two other types. If a block's original data is read by a load instruction, the block is called *load unmodified*. And a block is *partially modified* if it is evicted from the cache with unmodified portion and is not load unmodified. The state of a block is affected by both program characteristics and cache parameters. For instance, many potential fully modified blocks are evicted partially modified because of their short lifetimes in the cache. Figure 1 illustrates the states and transitions of store allocated blocks, and a newly allocated block is partially modified initially.

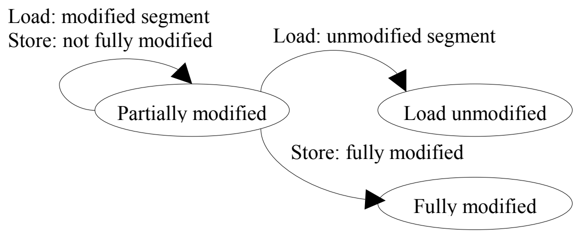

Figure 1. States and transitions of store allocated blocks. Store allocated blocks are allocated due to store misses.

This work makes two contributions. 1) We demonstrate the potential to effectively reduce both memory traffic and cache misses by directly installing fully modified blocks in data caches. For a 1MB data cache, 28% misses that access memory are avoidable. 2) We introduce a hardware mechanism, the Store Fill Buffer, which can efficiently identify avoidable misses by delaying fetches for store misses. The Store Fill Buffer has certain advantages over schemes such as write-validate caches [9] and cache installation instructions [8,15]. For example, the Store Fill Buffer requires no compile-time support and incurs minimal hardware overhead. For a 64KB data cache with a 16-entry Store Fill Buffer, 16% of overall data cache misses are eliminated, which results in 6% performance speedup across SPEC CINT 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 memory 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 memory traffic. One of such schemes is the write-validate cache [9], in which store allocated blocks are not fetched. Instead, the data is written directly into the cache, and extra valid bits are required to indicate the valid (i.e. modified) portion of the blocks. One of write-validate's deficiencies is the significant implementation overhead, especially when per-byte valid bits are required (e.g. Alpha ISA [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, which may negate write-validate's traffic advantage. As a comparison, the Store Fill Buffer reduces both load and store misses, and incurs far less overhead to yield comparable cache performance to a write-validate cache.

Write caches [9] are used with write-though caches to coalesce missed stores before they are written to lower levels of memory. Although both write-allocate or non-write-allocate write-miss policies can be utilized, write-allocate leads to better performance by exploiting the temporal locality of recently written data. And with a write-allocate cache, a write cache is unable to minimize avoidable memory traffic identified in this work.

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. For example, 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. The use of the instruction is further limited by the compiler's limited scope since it cannot identify all memory initialization instructions.

Recently, a hardware mechanism is proposed to identify stores that initialize heap objects, and trigger cache installation instructions to reduce memory traffic dynamically [11]. The mechanism's dependence on the `malloc()` system routine limits its application to programs that use the routine exclusively. For instance, the approach cannot work efficiently on Java programs since most Java virtual machines manage the heap by themselves. Furthermore, the mechanism can only identify heap object initializations, and leaves out fully modified blocks arising from other program activities. In contrast, the Store Fill Buffer identifies almost all fully modified blocks with no software assistance, and hence is valid for programs written in any languages.

3. Methodology

This section summarizes the time-accurate, execution-driven simulation environment and the SPEC CINT 2000 benchmarks used in this research.

3.1. Simulator

This work utilizes the revised Simplescalar/Alpha version 3.0 toolset [4] to obtain the characteristics of fully modified blocks and evaluate the performance impact of the Store Fill Buffer. Simplescalar/Alpha includes a suite of functional and timing 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 instruction and data caches. The configuration parameters of the simulated system are given in Table 1.

Table 1. Baseline configuration of the simulated machine.

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

3.2. Benchmarks

To perform our evaluation, we collect results from SPEC CINT 2000 benchmarks [16]. The benchmarks are compiled with SPEC peak settings, which performs 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% overall misses are store misses for a 64KB L1 data cache.

Table 2. Characteristics of SPEC CINT 2000 benchmarks. (Cache configurations: 64KB, 4-way, 64B blocks)

Benchmarks	Input set	L1 data cache miss rate	% of store misses in overall L1 misses
gzip	log	1.38%	26.57%
vpr	route	2.70%	15.76%
gcc	166	6.61%	52.74%
mcf	ref	18.61%	23.02%
crafty	ref	1.31%	12.55%
parser	ref	2.07%	10.38%
eon	cook	2.02%	31.52%
perlbmk	diffmail	0.78%	16.36%
gap	ref	4.43%	25.01%
vortex	two	1.22%	14.70%
bzip2	program	2.00%	27.81%
twolf	ref	5.48%	17.70%

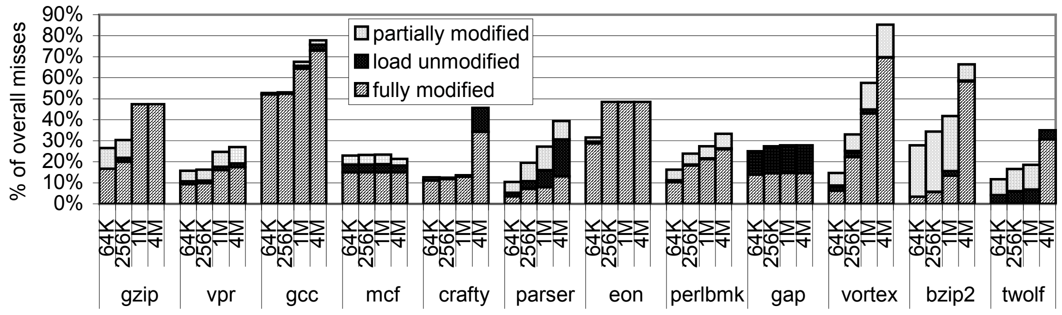

Figure 2. Cache miss breakdown. (64KB-4M caches, 4-way, 64B blocks)

4. Analysis of avoidable memory traffic

In this section, we demonstrate that large amount of memory traffic are avoidable, regardless of varying cache configurations. We also classify data references by their access types and the accessed block types. Finally, we study the stability of fully modified blocks by analyzing the fill intervals of those blocks.

4.1. Avoidable memory traffic

Figure 2 gives the percentages of the three types of store allocated blocks for write-allocate caches ranging from 64KB to 4MB. It is assumed that the two smaller caches represent the L1 data caches, while the two larger caches represent the total capacities of on-chip caches. Hence, the results show the data traffic that can be avoided either between L1 and L2 caches, or between L2 cache and memory.

The data traffic allocating fully modified blocks is avoidable. Since cache blocks stay longer in a larger cache, many otherwise partially modified blocks become fully modified in a larger cache. Consequently, the proportions of avoidable memory traffic increase with larger caches. On average, 28% misses of a 1M cache with 64B blocks allocate fully modified blocks. All data traffic for these misses can be eliminated since the allocated data will never be used.

Load unmodified blocks represent the increased load misses of a non-write-allocate cache over a write-allocate cache. Such load misses occur in a non-write-allocate cache when the invalid block portions are accessed. The corresponding increase of load misses is one of the reasons that write-allocate caches outperform non-write-allocate caches. Figure 2 shows that many programs have ignorable load unmodified blocks. One distinct program is gap, 11% of allocated cache blocks are load unmodified. Hence, a non-write-allocate cache will perform badly in gap.

Figure 3 illustrates the sensitivity of the three types of store allocated blocks to cache block sizes. As cache block size increases, the proportions of store misses drop. This demonstrates that stores have better spatial locality than loads. As cache blocks become wider, a store allocated block has a higher probability to be partially modified or load unmodified. As a result, the fraction of fully modified blocks decreases with wider cache blocks. However, even with wide blocks, there are still plenty avoidable memory traffic. On average 16% memory traffic are avoidable for a 1MB cache with 256B blocks.

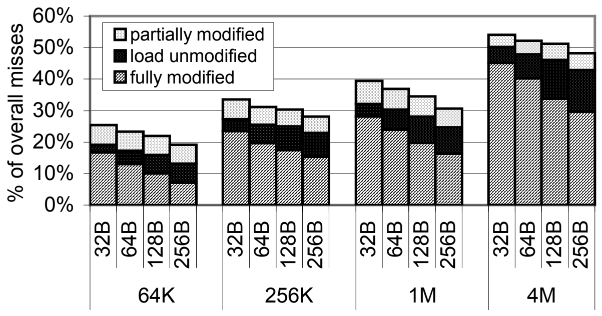

Figure 3. Sensitivity of avoidable memory traffic to cache size and block size.

Reducing avoidable memory traffic by directly installing fully modified blocks can improve performance by reducing pressure on store queues and cache hierarchies. In addition, eliminating avoidable memory traffic decreases memory bandwidth requirements. As a result, performance can be further improved by utilizing optimizations, such as prefetching and multithreading, that demand high memory bandwidth.

4.2. Decomposition of data references

In a write-allocate cache, cache blocks can be classified into load or store allocated blocks by their initiating miss types, and loads and stores may access both types of blocks. Figure 4 breaks down the data references by their reference types and accessing block types. 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 allocated blocks dominate most SPEC CINT 2000 benchmarks. On average, 66% of data references hit load allocated blocks, 30% of them hit store allocated blocks, and the other 4% miss in the data cache.

Interestingly, on average 18% of overall data references are loads hitting in store allocated blocks. More loads of gap access store allocated blocks instead of load allocated blocks, which partially accounts for the high percentage of load unmodified blocks in gap. The results also imply that by buffering store and load allocated blocks in separate caches, the conflicts between those two types of blocks may be effectively eliminated. Consequently, load misses can be reduced by structures such as the Store Fill Buffer, which is introduced in the next section.

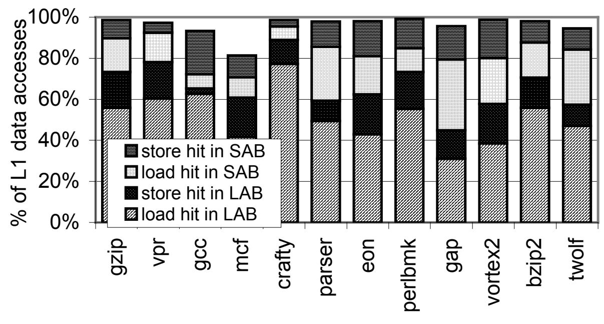

Figure 4. Breakdown of L1 data references. (LAB: load allocated blocks; SAB: store allocated blocks)

4.3. Fill intervals

Figure 5 categorizes fully modify blocks by the lengths of their fill intervals. A block's fill interval is the number of data references/stores executed during the period that the whole block is overwritten. A block with long fill intervals has a higher probability to be partially modified in case that its lifetime is short. Hence, the lengths of fill intervals reveal the stability of fully modified blocks. For benchmarks such as gzip, gcc and parser, most of the 64-byte blocks are filled by at most 16 stores. This implies that those blocks are filled up by series of successive stores without accessing other blocks. Heap object initialization is one source of the stores with such good spatial locality [11].

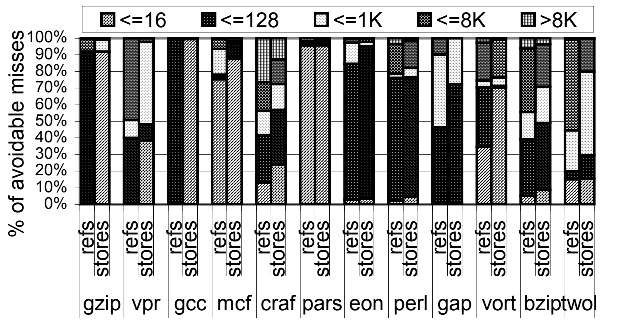

Figure 5. Intervals between allocation and fillness of fully modified blocks. (64KB data cache, 4-way, 64B blocks)

5. Eliminating avoidable memory traffic

To eliminate avoidable memory traffic, we must be able to identify fully modified blocks and install them directly in the data cache. We propose a hardware mechanism, the Store Fill Buffer, to temporarily buffer all store allocated blocks, and identify fully modified blocks in the mean time to reduce memory traffic.

In this work, the Store Fill Buffer assists the L1 data cache to reduce traffic between the L1 and L2 caches. However, there is virtually no limit to apply the idea to the L2 cache to further eliminate memory 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. It has the same block size as the L1 data cache. Considering its small size, the SFB hardly affects the L1 data cache's access latency. By default, the SFB 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.

When store misses occur, the requesting blocks are not fetched. Instead, all store allocated blocks are installed directly in the SFB and their modification states are monitored. As soon as a block's modification status, i.e. fully modified or load unmodified, is identified, the block is evicted to the L1 data cache. By doing so, the SFB makes the best use of its limited size. With a full SFB, the partially modified block in the LRU entry is evicted to the L1 data cache, leaving the SFB entry for the new block.

By employing the SFB, data traffic between L1 and L2 caches is reduced since writing fully modified blocks to the L1 cache incurs no fetches to lower levels of memory. For the non-fully modified blocks, the SFB delays the fetches of their original data until they are evicted from the SFB. Consequently, many load allocated blocks have longer lifetimes than in the baseline configuration. Hence, load misses incurred by the conflicts between load and store allocated blocks are reduced, which effectively offset the load misses increased due to invalid potion accesses.

The transfers between SFB and L1 cache is transparent to lower levels of memory hierarchies, and the L1 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 blocks, the performance penalty of a full SFB can also be minimized.

The SBF 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

In this subsection, we analyze the performance impact of the SFB and compare the SFB with the write-validate cache and the victim cache.

5.2.1 Avoidable memory traffic reduction

Figure 6 illustrates the overall data misses reduced by incorporating the SFBs with 16, 32 or 64 entries to the baseline 64KB write-allocate data cache. The two columns of each result bar represent the percentages of load and store misses eliminated by the SFB respectively.

A small SFB is effective on eliminating store misses. In a write-allocate cache, many partially modified blocks are potentially fully modified given long enough lifetimes. Since load misses dominate most programs, many store allocated blocks have similar lifetimes in the SFB as in the L1 cache of the baseline system. For most programs, nearly all fully modified blocks of a 64KB cache (Figure 2) can be recognized by a 16-entry SFB because of those 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.

Despite that accessing invalid portions of SFB blocks increases load misses, overall load misses are reduced by the SFB for programs such as mcf, crafty, eon, perlbmk, and vortex. By using the SFB, store allocated blocks are initially buffered in the SFB instead of the L1 data cache. Consequently, load allocated blocks stay relatively longer in the L1 data cache, and many conflict misses between load and store allocated blocks are avoided.

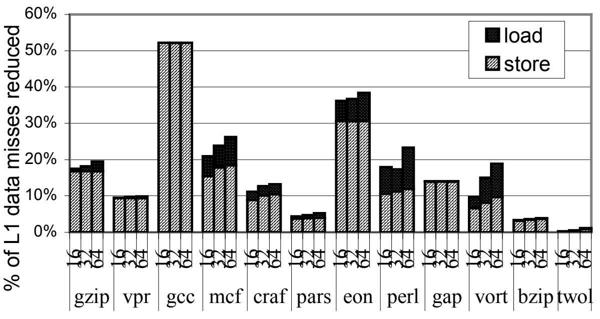

Figure 6. Percentages of L1 data misses reduced by Store Filled Buffers over a write-allocate cache. (64KB cache, 4-way, 64B blocks; SFB entries: 16, 32, and 64)

5.2.2 Comparison with other schemes

Although the SFB is similar to a write-validate cache, write-allocate with SFB is superior to write-validate in three aspects. First, a SFB incur less hardware overhead and uses on-chip transistors more efficiently than write-validate. For example, a 64KB write-validate cache needs additional 8K bytes to store block valid information. However, an 8KB SFB with 64B blocks has only about 862 bytes valid overhead, and the majority of the transistors can be used for buffering real data.

Second, the SFB scheme simplifies the memory interface design and implements partial block stores more efficiently. The write validate policy requires that the lower levels of memory support partial block writes. This may complicate the memory interface design and reduce bus efficiency. For instance, storing a partially modified block may require multiple bus transactions when several valid and invalid portions of the block intertwine with each other. In the SFB scheme, the partial block writes occurs on-chip, and can be implemented more efficiently and consume less time to fulfill. Furthermore, the partial block stores are totally transparent to lower levels of memory, which simplifies the design of lower memory hierarchies.

Finally, a SFB reduces load misses as well as store misses, as can be seen in Figure 6. In contrast, a write-validate cache reduces store misses at the expense of increased load misses arising from reading invalid portions of directly allocated blocks. Since the 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.

To justify the increased cache capacity by the SFB, we also compare the SFB with the victim cache [10]. Among all kinds of cache assists, victim cache is one of the most popular schemes. Victim cache is a small, fully set associative buffer holding discarded cache blocks. It is checked on cache misses to see if it contains desired data before going down to next level of memory hierarchy. Hence, it is effective on eliminating conflict misses.

Figure 7 shows the comparison results. For each scheme, the figure shows the percentages of L1 data misses reduced over the baseline 64KB write-allocate cache. Besides the additional 8KB valid bits, the write-validate cache has the same configuration as the write-allocate cache. Both the SFB and the victim cache have 16 entries. With significantly less overhead, the SFB outperforms the write-validate policy in half programs. It also exceeds the victim cache in eight programs.

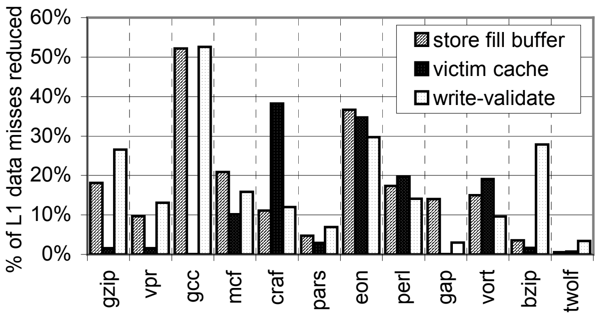

Figure 7. Comparison of Store Fill Buffer with write-validate cache and victim cache. (64KB caches, 4-way, 64B blocks; 16-entry Store Fill Buffer and 16-entry Victim Cache)

5.2.3 Performance impact

Figure 8 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, more than 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 rate and abundance of avoidable misses. On the other hand, the SFB has almost no impact on the performance of perl and twolf.

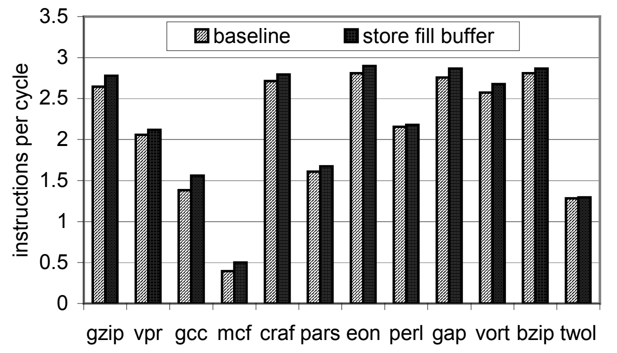

Figure 8. Performance speedup by the Store Fill Buffer. (Baseline configurations as shown in Table 1; 16-entry Store Fill Buffer)

6. Conclusions

Memory bandwidth limitation will be one of the major impediments to future microprocessors. Hence, reducing memory bandwidth requirements 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, those blocks can be directly installed in the cache to reduce memory traffic. The amount of fully modified blocks is affected by program characteristics and cache parameters. For the SPEC CINT 2000 programs, on average 28% 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 memory traffic. By delaying fetches for store misses, 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. By incurring significant less overhead, the Store Fill Buffer provides comparable performance to a write-validate cache. The Store Fill Buffer is also superior to the victim cache in cache performance. 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 CINT 2000 benchmarks.

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.