

Write Buffer Design for Cache-Coherent Shared-Memory Multiprocessors

Farnaz Mounes-Toussi and David J. Lilja

Department of Electrical Engineering
University of Minnesota
Minneapolis, MN 55455

Abstract

We evaluate the performance impact of two different write-buffer configurations (one word per buffer entry and one block per buffer entry) and two different write-policies (write-through and write-back), when using the partial block invalidation coherence mechanism [3] in a shared-memory multiprocessor. Using an execution-driven simulator, we find that the one word per entry buffer configuration with a write-back policy is preferred for small write-buffer sizes when both buffers have an equal number of data words, and when they have equal hardware cost. Furthermore, when partial block invalidation is supported, we find that a write-through policy is preferred over a write-back policy due to its simpler cache hit detection mechanism, its elimination of write-back transactions, and its competitive performance when the write-buffer is relatively large.

Keywords: cache coherence, write-buffer, write-merging, write-through, shared-memory multiprocessors

1 Introduction

One of the important issues in designing a cache memory is to decide what happens during a write operation. There are two basic write policies: *write-through* and *write-back*. With a write-through policy, both the cache and the memory are updated during every write operation. Hence, when the cache overflows or a conflict miss occurs, the block can be overwritten with no additional processor-memory transactions. With a write-back cache, however, only the cache is updated during a write operation. Thus, the block contents must be written back to the memory before the block is replaced.

In addition to the processor-memory traffic overhead from block replacements, a write-back cache introduces the *ownership overhead* in a shared-memory multiprocessor. With a write-back cache the memory copy may be stale, that is, the value in memory may be an old version with respect to the cached copy. In addition, other processors in the system may also have stale copies. Consequently, some mechanism, such as ownership, is required to keep track of the processor with the most up-to-date copy of the block. An

ownership-based coherence protocol forces a processor to request exclusive access to a block before it writes to a block that it does not own. This ownership transfer is accomplished by requesting that the block be written-back to memory by the current owner if the block is already modified (and, therefore, owned) by some other processor, or by invalidating all other valid copies of the block if it is currently shared.

With a write-through cache, however, the memory copy of the block is always up-to-date (within the limits defined by the *consistency model* [5, 7]) so that no ownership transfer or write-back is necessary when any processor writes to a block. The writing processor simply invalidates other cached copies to prevent future accesses to stale data and writes the new value to both the cache and the memory.

While the write-through cache eliminates the processor-memory transactions due to block replacements and ownership, it introduces a write-through transaction for every write operation. Thus, the processor-memory communication overhead may increase substantially due to the frequent write-throughs. Jouppi [8] has shown, however, that in a uniprocessor, a small write-buffer can be quite effective in reducing the write-through traffic. The write-buffer is associatively searched during every write operation so that multiple writes to the same memory location can be merged. As a result, only one write operation is eventually propagated to the memory. In addition, a study by Chu and Gottipati [4] shows that a write-buffer configured as a FIFO queue (i.e. a buffer with no write-merging capability) can reduce the program execution time in a uniprocessor with a write-through cache compared to the execution time with a write-back cache.

While these two previous studies focus on the uniprocessor environment and assume one or two words per buffer entry, Bianchini et al [1] investigate the effects of using an associatively searched write-buffer with one block per entry (i.e. each entry corresponds to an entire cache block) in a distributed shared-memory multiprocessor. Their study shows that this write-buffer improves performance in the presence of false sharing [6]. Our previous study [9] comparing write-through and write-back caches with partial block invalidation in shared-memory multipro-

processors showed that a write-through cache can outperform a write-back cache when using a write-buffer. These two studies confirm the effectiveness of merging writes with an associatively searched write-buffer in a multiprocessor environment.

While the uniprocessor write-buffer studies [4, 8] assume one or two words per buffer entry, the multiprocessor studies [1, 9] use write-buffers in which each entry corresponds to an entire cache block. This one block per entry configuration, however, may not use the available hardware as efficiently as the one word per entry configuration since not all of the words in a cache block typically will be written between flushes of the buffer. On the other hand, potentially better utilization in the one word per entry configuration comes at the expense of additional hardware to store the word address as well as the block address, and the additional communication overhead of sending separate messages for each word in the same block rather than one message for the entire block. In this paper, we investigate the performance impact of these two write-buffer configurations with both write-through and write-back policies in a shared-memory multiprocessor with partial block invalidation. These two write-buffer configurations are compared with respect to the total number of data words in the buffer and the estimated hardware implementation cost.

The remainder of the paper is organized as follows. Section 2 describes the operation of the partially valid coherence schemes. Section 3 describes the estimate of the hardware cost. Section 4 describes the simulation methodology. Section 5 presents the simulation results and Section 6 summarizes our findings.

2 Partial block invalidation coherence mechanisms

In a shared-memory multiprocessor with private data caches, modifying a cached block with multiple copies in the other caches introduces the *cache coherence problem*. To ensure consistency among multiple copies of the same memory location, a coherence scheme, such as the ones described below, is required.

Partially valid write-back mechanism In the partially-valid write-back scheme proposed by Chen and Dubois [3], each cache block is divided into two or more *invalidation blocks*, each with its own *valid* bit. A *dirty* bit is also associated with the entire cache block. A block is owned by a processor when the single *dirty* bit for the block is set in that processor's cache. The individual *valid* bits are used to keep track of those invalidation blocks that have been modified. When the block is owned and the *valid* bit of the invalidation block is set, all read and write requests are performed locally. However, when the *valid* bit is not set, other processors may also have a valid copy of this invalidation block. In this case, a writing processor must send a request to invalidate the other cached copies of this invalidation block. If the block is not owned (i.e. the *dirty* bit is not set), the *valid* bits keep track of which invalidation blocks have valid data. Furthermore, in the memory, each invalidation block is associated with

a P -bit vector, where P is the number of processors in the system. Each memory block also has a pointer to the owner of the block to indicate which processor last wrote to the block, if any. Using the combinations of the *valid* bits and the *dirty* bit, a cached block may be in any of the following three states:

- **Shared:** the *dirty* bit is reset and all valid bits are set. The entire cache block is valid and other copies of this block may exist in other caches. A read reference to a Shared block generates a cache hit, while a write requires invalidation of all the valid copies of the corresponding invalidation block before writing to the cache. This write reference resets all of the valid bits except the valid bit of the invalidation block that is modified, and sets the dirty bit of the writing processor's cache block.
- **Owned:** the *dirty* bit is set. The entire cache block is valid, but other copies of the invalidation block may exist in other caches if the *valid* bit of the invalidation block is not set (i.e. the invalidation block is being written to for the first time since ownership was granted). A read reference to an owned block generates a cache hit, but a write to an invalidation block with its valid bit reset requires invalidation in the other caches with a valid copy of the invalidation block in the writing processor's cache. This write reference sets the valid bit of the modified invalidation block.
- **Stale:** the *dirty* bit is reset and at least one of the valid bits is reset. This state implies that some other processor has written to this block. As a result, a write to this block will generate a write miss request which causes the entire cache block to be written-back to the memory before the write miss request is serviced. A read reference to a Stale block generates a miss only if the invalidation block is invalid. Otherwise, it will generate a hit.

Note that when an owned block is replaced, the current contents of the block must be written-back to the memory before reallocating the cache block. Also, to detect a write hit or miss, all of the valid bits in a block must be checked.

Partially valid write-through mechanism Our partially valid write-through scheme [9] also divides each cache block into invalidation blocks and assigns one *valid* bit per invalidation block. The memory directory uses one P -bit vector per invalidation block to keep track of the processors with a valid copy of each invalidation block. Given the two possible states of a *valid* bit, an invalidation block may be in either of the following two states:

- **Shared:** the invalidation block is valid in the cache. Other processors may also have a valid copy of this invalidation block. A write reference to a Shared invalidation block requires the invalidation of other copies of the invalidation block

and the forwarding of the new data to the memory. This write reference will set the valid bit of the corresponding invalidation block. A read reference to a Shared invalidation block is performed locally, however.

- Invalid: the invalidation block is not valid in the cache so that the entire block must be fetched from the memory. If the miss is due to a write reference, other copies of the invalidation block must be invalidated and a write-through of the new data to the memory must be performed. A read or write miss request sets all of the valid bits in the block.

Compared to the partially valid write-back cache, the partially valid write-through cache simplifies the cache hit detection mechanism by needing to check only one of the valid bits in the block. However, since the write-through cache is required to forward data to the memory in every write operation, we can expect higher communication overhead with a write-through cache than with a write-back cache. Nevertheless, in a shared-memory multiprocessor which supports weak consistency [5, 7], a write-buffer can be used to reduce the communication overhead of a write-through cache [9]. The write-buffer reduces processor stalls on write misses, and on write hits to an unowned block, by allowing multiple outstanding write requests. The outstanding requests are forwarded to the memory while the processor is busy performing other operations. Furthermore, to allow merging of writes to the same memory location, this forwarding can be delayed until a synchronization point is reached instead of immediately forwarding the outstanding requests.

3 Write buffer implementation

Assuming a block size of 16 words and an 8 entry write-buffer, the one block per entry configuration associates one address tag with each cache block of 16 words. Since there are 8 tag entries, this configuration has 128 total words of storage. The one word per entry buffer configuration, on the other hand, associates one address with *each word*. Consequently, two words from the same block will consume two tag entries in this configuration, while they would consume only a single entry in the other configuration. Also, since each word must be uniquely identified in the one word per entry configuration, its address tags will require more bits than the one block per entry configuration.

During a write operation, both the processor's cache and its write-buffer are associatively searched in parallel. If the block or the word being written is already in the buffer, the new value simply overwrites the old value, thereby effectively merging write operations to the same block or word. If the desired word or block is not already in the buffer, and the buffer is not full, a new entry is allocated. The data word then is simultaneously written to the buffer, its corresponding valid bit is set, and the type of request is stored. If, however, the buffer is full (i.e. the buffer overflows), the contents of the entire buffer are flushed by writing all the modified words to memory. The processor is stalled during this buffer flush operation.

There are two types of outstanding requests, *write-miss* and *write-hit*, which will be denoted as *write-miss-wt* and *write-hit-wt* for the write-through cache, and *write-miss-wb* and *write-hit-wb* for the write-back cache. Note that the type of request is known at the end of the data cache and write-buffer search. In response to a *write-miss-wb* or *write-miss-wt* request, the memory sends a copy of the entire cache block to the requester. A processor that generates a *write-hit* request, on the other hand, receives an acknowledgement from the memory which will change the state of the cache block to *owned*, if it is not already, and will set the valid bit corresponding to the modified word, if the write-back policy is used. It should be noted that with the *write-miss-wt* and *write-hit-wt* requests only the modified words are forwarded to the memory. The reception of an acknowledgement or of a requested cache block initiates a write-buffer search and deallocation of the corresponding buffer entry.

To approximate the hardware cost of the two write-buffer configurations, we used the Magic layout tool to implement the basic cells, to generate the Spice code for design verification, and to obtain an approximate transistor count. The cost approximation is based on the total number of transistors required to store data (DataRAM), address tags (TagRAM), and the status bits (i.e. the valid bit and the type of request bit). The TagRAM cells are implemented as CAM cells (9 transistors per CAM cell) to allow an associative search of the write-buffer. The status bits and the DataRAM, however, are implemented using standard SRAM cells (6 transistors per SRAM cell). Note that the cost of row address decoders in the TagRAM and DataRAM varies depending on the number of entries in the write-buffer. Also, both the TagRAM and the DataRAM require write drivers (6 transistors per cell) and sense amplifiers (5 transistors per cell). Table 1 summarizes the approximate cost of a write-buffer with one cache block (16 words) per entry and 24-bit address tags.

Table 1: Approximate hardware cost for a write-buffer with one block (16 words) per entry and 24-bit address tags.

# entries	Total # words	# transistors
1	16	11,030
4	64	28,075
16	256	102,276

To approximate the number of entries in a one word per entry write-buffer configuration when using the same number of transistors as the one block per entry configuration, we used multiples of the total number of transistors required to implement an 8-by-1-word write-buffer, which is approximately 6282 transistors. The write-buffer sizes for the two configurations when both are implemented using the same number of transistors are: 16, 64, and 256 words with the one block per entry configuration, which corresponds to 16, 40, and 128 words with the one word per entry configuration, respectively. We also studied the effect of 32 and

128 words buffer sizes, but they showed a behavior similar to the ones presented in Section 5.

4 Simulation methodology

Using an execution-driven simulator with 16 processors, we compare the relative effectiveness of write-buffers with one word per entry to buffers with one block (16 words) per entry, using partially valid write-through and write-back caches. These simulations assume a direct-mapped 16 Kbyte (4 Kword) cache with a block size of 64 bytes (16 words). Our simulations have shown that a 16Kbyte cache is sufficient for the working set of the test programs used in this study. Both the write-through and the write-back caches allocate a block on a write miss and stall the processor on a read miss. If, on a read miss, the requested block is found in the write-buffer, the write-buffer is flushed back to memory. The read operation is performed after the buffer flushing is completed.

The simulator is tightly coupled to the MINT [11] reference generator which models the execution of an application program on some number of processors, where each processor requires 1 cycle per instruction. Our architecture simulator models a shared-memory multiprocessor where the processors and the memory modules are connected through a general interconnection network [7]. The invalidation block size is assumed to be 1 word (4 bytes).

All accesses to the cache, and all writes to the write-buffer (in the absence of a buffer overflow), require 1 cycle. Each buffer entry contains the data to be written, one valid bit per word, the type of outstanding request, and the address of the block, or the address of the word within the block, depending on the buffer configuration. While a request is outstanding in the write-buffer, the cache block may be replaced before the request has been serviced. To ensure that the written value will not be lost when the block is replaced, and to eliminate the unnecessary transfer of a missed block which is already replaced in the processor’s cache, the type of request is changed to a *replaced-wt* request. When the memory receives this request, it invalidates other valid copies of the invalidation blocks which contain the modified word(s) and sends a *replaced-ack* message to acknowledge the processor’s request.

Figure 1 shows how the type of outstanding requests may be changed as they go through a three stage pipeline during a buffer flush. In the first stage, data, address, and status bits are read from the buffer. In the second stage, the address tag of the buffer entry just read is compared with the address tag stored in the cache. This comparison is used to change the type of request to a *replaced-wt* transaction. The memory sends an acknowledgment to the processor upon completion of the transaction which then deallocates the corresponding write-buffer entry. The last stage of the pipeline updates the type of request and forwards it to the memory.

The latency of a memory access through the network depends on the type of request, the state of the memory block, and memory contention. A processor executing a memory operation that produces a miss

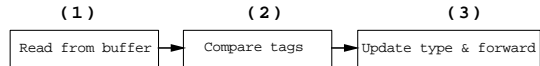


Figure 1: The three stage pipeline for changing the type of an outstanding request during a write-buffer flush.

or a buffer flush is blocked until the memory operation is completed. Table 2 shows the latency for servicing each type of memory access, with no memory contention and no network contention, and a memory access time of 5 cycles. The simulator adds additional delays to the memory requests that are issued by different processors to simulate the effects of memory contention.

Table 2: Latencies for different requests through the network.

Processor-to-Memory	Latency in processor cycles
write-miss-wb request	15
write-miss-wt request	15 + W
read-miss request	15
write-hit-wb request	15
write-hit-wt request	15 + W
replaced-wt	15 + W
write-back	15 + B
acknowledgment	15
Memory-to-Processor	
write-back	15
invalidate	15
miss service	15 + B
acknowledgment	15
replaced-ack	15

W = number of modified words
B = cache block size (in words)

We use the normalized total execution time, measured in processor cycles, to compare how the different write strategies and buffer configurations affect the system performance. All of the results are normalized to the performance of the SC scheme which is a directory-based write-back scheme [2] with no write-buffer and a cache block size of 1 word.

Three test programs from the Splash benchmarks [10] and one test program from the SPEC-1992 benchmarks are used for this study. *Barnes-Hut* simulates the evolution of 1024 bodies under the influence of gravitational forces. *Mp3d* simulates the flow of 3000 molecules through a rectangular tunnel for 50 time steps. *Cholesky* performs a Cholesky factorization of a 1806-by-1806 sparse matrix. The program *Vpenta* from the SPEC-1992 benchmarks inverts three matrix pentadiagonals. These test programs are parallelized by hand and conform to a *fork-join* programming model. As shown in Table 3, more than 34 percent of the total references of *Mp3d* and *Barnes* are writes, while for *Cholesky* and *Vpenta* fewer than 24 percent of the total references are writes. Also, *Cholesky* synchronizes more than twice as much as the other programs. Frequent synchronization can reduce the effectiveness of write-merging since the weak-consistency

model requires the write-buffer to be flushed at each synchronization point. Fewer write operations implies that the performance of the program is less sensitive to the write-buffer size than a program with many write references.

Table 3: General characteristics of the test programs.

Program	Total references	%write operations	# synch. ops. (X 1000)
Vpenta	544891	22.23	6.61
Mp3d	9318045	34.95	23.48
Cholesky	28085300	23.45	49.16
Barnes	58034340	37.05	24.84

5 Simulation results

This section compares the performance of the following write-buffer and write policy configurations: write-back with write-buffer, 16 words/entry (WBB); write-back with write-buffer, 1 word/entry (WBW); write-through with write-buffer, 16 words/entry (WTB); and write-through with write-buffer (WTW), 1 word/entry, where each entry corresponds to the address tag, status bits, and data word.

5.1 Equal data size configuration

Figure 2 shows the impact on execution time on the left, and the total communication overhead on the right, for the four different configurations when using the same total number of words in each write-buffer. For *Barnes* and *Mp3d*, the high communication overhead degrades the performance of the WBB scheme compared to the WBW scheme, when the buffer size is 16 words. For a buffer size of 64 or greater, however, the WBB scheme is competitive with the WBW scheme due to the 40 to 60 percent reduction in communication overhead with a buffer of 64 or 256 words. This reduction is due primarily to fewer write-buffer flushes, and due to the sending of one message for every 16 words rather than one message for every word in the buffer (as is done in the WBW configuration) during a buffer flush. For *Cholesky* and *Vpenta*, on the other hand, both configurations are competitive regardless of the buffer size, since these two programs perform fewer write operations than *Mp3d* and *Barnes*.

A comparison of the WTB and WTW configurations shows that with a small (e.g. 16 words) buffer the high communication overhead degrades the performance of the WTB configuration for *Barnes*, *Vpenta* and *Mp3d*. A write-buffer of 256 words, however, reduces the communication overhead of the WTB configuration to below that generated by the WTW configuration. For *Cholesky*, again, both buffer configurations are competitive for a given buffer size, but the total execution time with the write-through cache reduces by about 15 percent when the buffer size increases from 16 to 64 words. Since a write-through cache writes to both the cache and the buffer every time a write operation is performed, its performance is

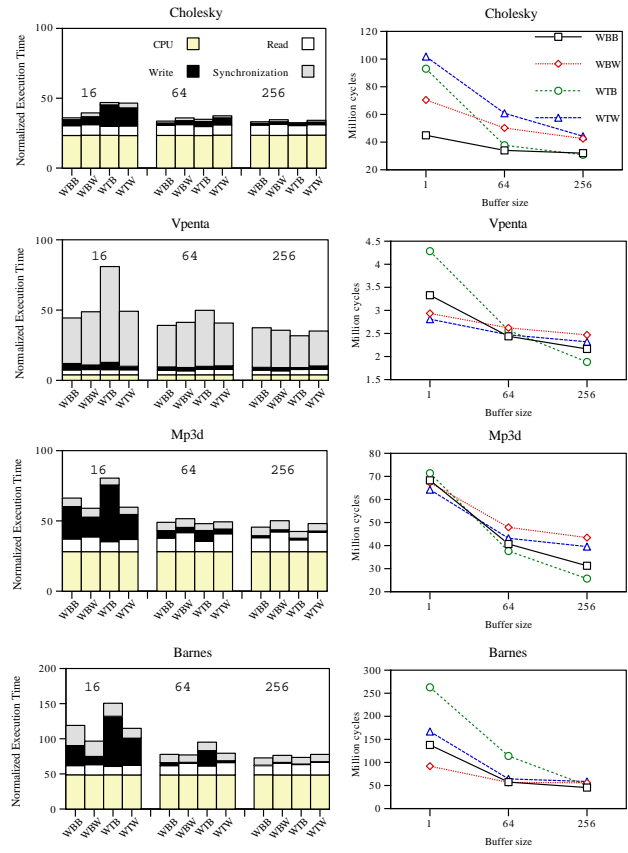


Figure 2: Normalized execution time and total communication overhead for write-buffers with 16, 64, and 256 total words.

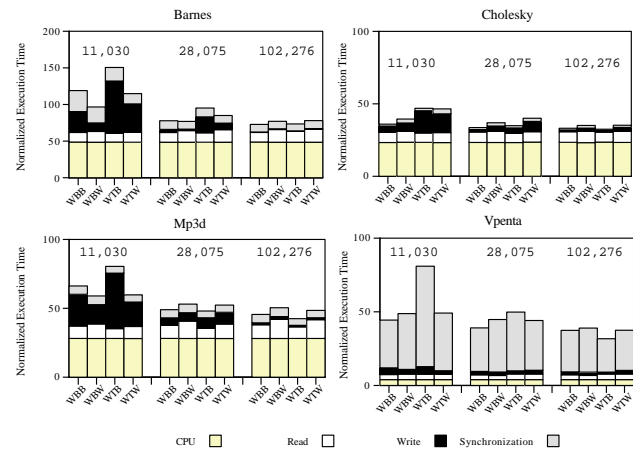


Figure 3: Normalized execution time when the two write-buffer configurations are implemented with 11,030, 28,075, and 102,276 transistors each.

more sensitive to the write-buffer size than the write-back cache.

A comparison of the write-through and write-back policies indicates that with a small write-buffer (e.g. 16 words) the write-back policy is preferred. With a large buffer (e.g. 256 words), however, the one block per entry configuration is preferred using either write-back or write-through. Nevertheless, given the fewer states, no write-back requests, and simpler cache hit detection mechanism when using the write-through policy, we suggest choosing the write-through with one block per entry (WTB) configuration over the write-back with one block per entry (WBB) configuration, if partial block invalidation is supported.

5.2 Equal cost configuration

The previous section indicated that when the write-buffers have an equal number of data words, the write-back policy and the one word per entry buffer configuration are preferable with a small write-buffer. However, the hardware cost of the one word per entry configuration is higher than the cost of the one block per entry configuration due to the need for one address tag with each data word rather than every 16 data words. Also, the size of the address tag increases when the buffer is implemented in the one word per entry configuration.

Figure 3 shows the performance of the two write-buffer configurations when implemented using approximately the same number of transistors. As shown in this figure, the behavior is essentially the same as that shown in Figure 2 where the configurations have the same number of data words. The only difference between these two figures is that the performance of the WTW and the WBW configurations in Figure 3 slightly degrades compared to the performance of the corresponding configurations shown in Figure 2. This degradation is due to the smaller total number of words in the one word per entry configuration than in the one block per entry configuration.

6 Conclusions

In this paper, we have investigated the use of one word per entry and one block per entry write-buffers in conjunction with partially valid write-through and write-back caches. Using an execution-driven simulator, we find that for an equal number of words, or for equal hardware cost, for each buffer configuration, the write buffer with one word per entry performs well when the buffer is small and the cache uses a write-back policy. With a large buffer, however, the one block per entry configuration performs better regardless of the write policy. In summary, we conclude that a cache-coherent shared-memory multiprocessor system should be constructed with a one block per entry merging write-buffer configuration using a write-through policy.

Acknowledgements

We would like to thank Jack Veenstra for access to the MINT simulator, and John Mejia for the parallelized SPEC benchmarks. This work was supported in part by National Science Foundation grant no.

CCR-9209458, and by a University of Minnesota McKnight Land-Grant Professorship.

References

- [1] R. Bianchini, T. J. LeBlanc, and J. E. Veenstra. Eliminating useless messages in write-update protocols on scalable multiprocessors. Technical Report 539, University of Rochester, November 1994. Department of Computer Science.
- [2] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache coherency schemes. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.
- [3] Y. Chen and M. Dubois. Cache protocol with partial block invalidation. *7th International Parallel Processing Symposium*, pages 16–23, 1993.
- [4] Pong P. Chu and Ramana Gottipati. Write buffer design for on-chip cache. *International Conference on Computer Design*, pages 311–316, 1994.
- [5] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *International Symposium on Computer Architecture*, pages 422–434, 1986.
- [6] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. *International Conference on Parallel Processing*, pages 377–381, 1991.
- [7] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.
- [8] N. P. Jouppi. Cache write policies and performance. *International Symposium on Computer Architecture*, pages 191–201, 1993.
- [9] F. Mounes-Toussi and D. J. Lilja. A write-through cache coherence mechanism with partially valid blocks and write-merging. High-Performance Parallel Computing Research Group Technical Report no. 94-14, University of Minnesota, 1994.
- [10] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.
- [11] J. E. Veenstra and R. J. Fowler. MINT tutorial and user manual. Technical Report 452, University of Rochester, June 1993. Department of Computer Science.