Interactions between Application Write Performance and Compilation Techniques: A Preliminary View

Margaret Martonosi
Dept. of Electrical Eng.
Princeton University
Princeton, NJ 08544

Kelly Shaw
Dept. of Computer Science
Duke University
Durham, NC 27708

Abstract

Although write buffers are included in nearly every current microprocessor architecture, little research has focused on understanding how their design interacts with common application memory referencing characteristics. This paper describes a preliminary view of the interactions between application write behavior and write buffer performance for two case study applications. We find that compiler optimizations have a larger relative impact on program performance after a write buffer is introduced into the system. Furthermore, our more detailed views of application behavior show that a sizable component of write-buffer-related stalls are due to procedure call and return sequences in these codes. Overall, this study's preliminary results motivate further study into how compiler techniques can minimize buffer stalls, particularly at procedure boundaries, in order to improve overall application performance.

1 Introduction

As processor speeds have increased faster than memory speeds, a number of tactics have been employed to help hide the processor-memory performance gap. For example, write buffers have been incorporated into the memory hierarchy of most modern processors. Write buffers have two primary goals: to reduce the program latency incurred on memory references and to reduce the traffic that continues on to subsequent levels of the memory hierarchy. While write buffers are almost universally accepted, only a handful of studies have quantified the tradeoffs involved with write buffer design choices [2, 6].

This paper looks at write behavior for two benchmarks, and ties performance details back to the code itself. In particular, we make a first step towards understanding how particular compiler details, such as register allocation and procedure call sequences, impact program write behavior and overall program performance. We find that compiler choices, particularly those regarding procedure call/return sequences, can have a large impact on write buffer stalls. Our preliminary observations suggest that compiler writers should give potential write buffer impact ample attention when designing and implementing optimizations.

2 The Basics of Write Policies and Write Buffer Design

Before we describe our evaluations, we will first give a brief discussion of the design parameters we have explored in this paper. Since we are interested in application write behavior, the cache's write miss policy is relevant. Jouppi has given an overview of different write miss policy combinations in [4]. In particular, there are three main decisions to be made, and these are somewhat (but not entirely) orthogonal to each other. First, we choose whether or not to allocate a cache line when a write miss occurs. Second, we choose whether to fetch the entire cache line when a write miss occurs. Third, we choose whether to allow a write-before-hit optimization. Different combinations of these three choices lead to four different write miss policies: fetch-on-write, write-validate, write-invalidate, or write-around.

Fetch-on-write (FOW) means that on a write we fetch the entire cache line corresponding to the address of the write into the cache. In write-validate (WV), the system allocates a spot in the cache for the accessed line, but does not fetch the line from memory. It writes only the changed data to the cache line, and uses sub-block valid bits to indicate that only part of the cache line is currently valid. In the write-invalidate policy (WI), we also do not fetch the accessed line. In this policy, the system does a write to the correspond-

ing cache line while simultaneously checking to see if the write is a cache hit or a cache miss. If the write is a cache hit, then the simultaneous write was legal. Otherwise, the cache line must be invalidated because we have just written to a cache line that is not related to the current write. In write-around (WA), we write directly out to the remaining memory hierarchy and do not write anything to the cache or evict any current cache lines.

Data that is written to a cache is also written to the write buffer. That data is held in the write buffer until it is written to the next level of the memory hierarchy. If another write occurs to data that is currently in the write buffer, this new information alters the data in the write buffer and results in only one write being sent to the next level of the memory hierarchy. This is called a coalescing hit. One complication with a write buffer is that it could fill up. This would force a "buffer full stall", in which any further writes are stalled until a slot in the buffer has been emptied. A second complication is that a load may occur at a point when the requested data is in the write buffer. In this case, we assume that the load is stalled until the write buffer has been flushed (load service stall). Thus there are tradeoffs between (a) keeping data in the write buffer longer to allow more coalescing hits versus (b) sending data quickly to the next level in order to minimize load service and buffer full stalls. These tradeoffs are affected by the "high water mark", which indicates how many entries the write buffer will contain before we begin to retire items from it. We consider different "high water marks" in this study.

3 Methodology

3.1 Simulation Platform

In order to examine the effects of different program, compiler, and architecture interactions, we have a constructed a memory system simulator using MINT [7]. MINT is a software package that provides an interface between programs (both sequential and parallel) and event-driven memory hierarchy simulators. MINT uses a mixture of direct execution techniques and software emulation to minimize the simulation overhead. In this study, we primarily focus on sequential application behavior, although our simulator and experiments could be extended to parallel execution in the future.

3.2 Architectural Parameters

Our simulator focuses on the data cache and write buffer effects of sequential programs. We assume ideal instruction cache behavior, and simulate only the data cache. The capacity and line size of this cache are parameterized, but by default are set to 8KB and 32 bytes respectively.

We have implemented a simulator which allows for all logical combinations of write-through vs. write-back caches with the four possible write miss policies. Our default cache is write-through and its write-miss policy is fetch-on-write.

Finally, our simulator allows different choices regarding the write buffer design and policies as well. The main architectural choices regarding the write buffer include its depth (number of entries) and width (size of each entry). Our default write buffer has 8 entries, each of which is a cache line wide.

The write buffer stalls either when it is full or when a load requests data currently in the write buffer. For this reason, there are policy decisions to be made including when to retire write buffer entries and how to service loads that request data currently in the write buffer. Our write buffer retirement policy is based upon the number of entries in it; when the number of entries in the write buffer is greater than or equal to the high water mark, the write buffer schedules the oldest entry to be written out to the next level of the memory hierarchy. The high water mark can be varied from one to the number of entries in the write buffer. Our load service policy is a simple one; we require the load to wait until the entire write buffer is flushed. Future work could examine the impact of other load service policies on performance and compiler interactions.

3.3 Applications Studied

Our study focuses on the memory behavior of two programs. The first of these is a sequential implementation of the Travelling Salesman Problem (TSP) from the Olden benchmark suite [1]. This implementation creates a binary tree to represent possible paths, and then uses a modified version of Karp's divide-and-conquer algorithm [5] to find a solution. The application has three main stages: tree-building, list-making, and conquer.

The second benchmark, xlisp, is a lisp interpreter solving the n-queens problem, and is one of the SPEC92 benchmarks [3]. The standard SPEC input is for solving the nine-queens problem. We have exam-

ined both the eight-queens and nine-queens problem sizes.

4 Application Memory Performance: An Overview

The main goal of this paper is to present application-specific performance details that note the interactions between compiler characteristics and architectural decisions. Before we present results at that detail, however, we will first give an overview of the general cache and write buffer behavior of the applications being studied.

Figure 1 gives an overview of the application overall performance. This graph shows data from the two applications, xlisp and tsp. For each application, the graph shows execution times for the four different write miss policies considered. Normalization is used so that the two applications, with their very different runtimes, can be shown on the same graph. The execution times are normalized to that for the fetch-on-write policy in both cases (These base times are 644 million simulation cycles for TSP and 37,500 million simulation cycles for xlisp.)

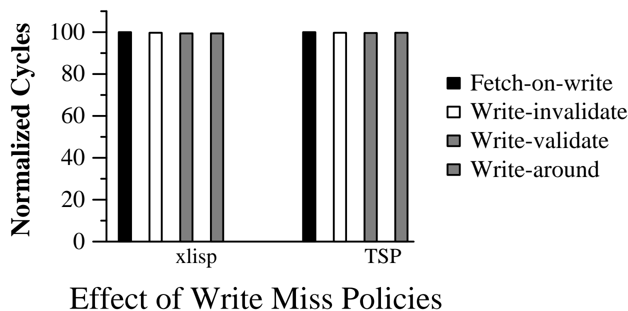


Figure 1: Effect of write miss policy in a system with no write buffer.

The addition of a write buffer into the system has a dramatic effect on the execution time. For xlisp, adding an 8-entry, cache-line-wide write buffer into the system reduces the execution time by roughly a factor of three. For tsp, the reduction is slightly larger, a factor of 3.7. Figure 2 shows a similar plot as Figure 1, except this time for a system including a write buffer of the size just described, with a high water mark of 4. The times in this plot are renormalized, to allow closer comparison between the different write policies. One can see that in addition to greatly improving application performance, write buffers also increase the sensitivity of application performance to a particular write policy. While the write policy makes essentially no difference in these applications when no write buffer is present, it can make a roughly 5% difference in the performance of the codes once a write buffer is added.

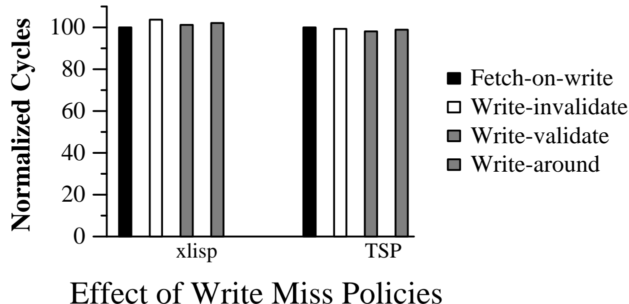


Figure 2: Effect of Write Miss Policy in a system with a write buffer. The write buffer has 8 entries, each a cache line wide, and has a high water mark of 4.

4.1 Write Buffer Parameters

Having presented some overview statistics that demonstrate the importance of write buffers on overall performance and their impact on other memory-related design choices, we now move on to other design choices that arise when considering write buffers. This section examines the impact of different high water marks on performance. Recall that the high water mark determines how full a write buffer will be before it begins to retire entries to memory.

As Figures 3 and 4 show, the number of buffer full and load service stalls increases dramatically as the high water mark increases. Increasing the high water mark means that we will not be retiring data out of the write buffer as quickly; thus, it is more likely that we will fill the buffer. Likewise, data is staying in the write buffer longer, so it is more likely that a load to the cache line will occur during its stay. Thus, we also expect more load service stalls.

Although increasing the high water mark increases the likelihood of buffer full and load service stalls, it also increases the likelihood that subsequent writes to the same line will result in references being coalesced. These coalescing hits in the write buffer can improve program performance by reducing the number of operations that continue on to subsequent levels of the memory hierarchy. Thus, they reduce contention for resources at those levels. As Figure 5 shows, coalescing hits in these two applications increase by about 20

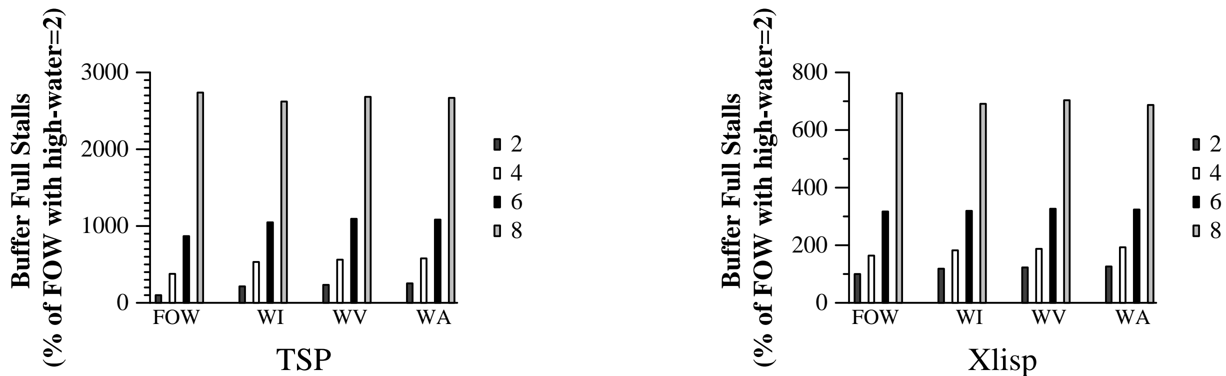


Figure 3: Frequency of write buffer stalls as a function of high water mark. A count of write buffer stalls is shown for each miss policy and high water mark. Each is normalized such that Fetch-on-Write with a high water mark of 4 is 100%.

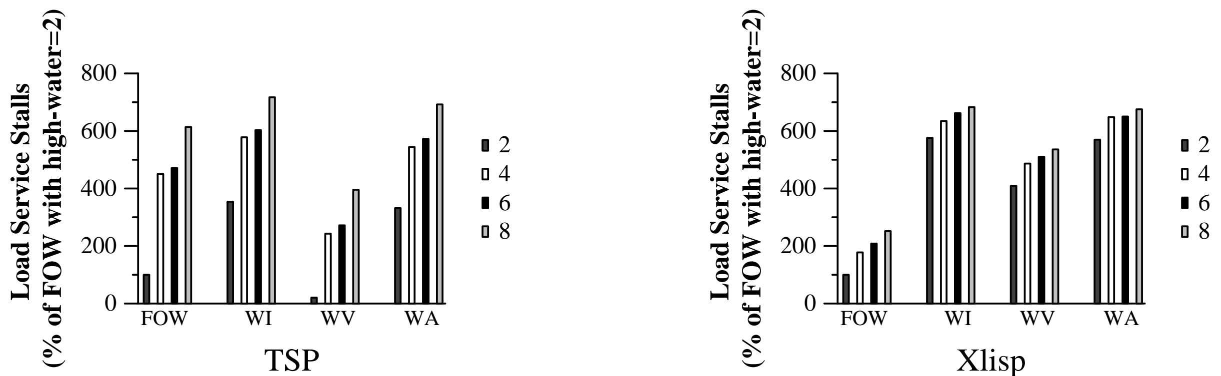


Figure 4: Frequency of load service stalls as a function of high water mark. A count of load service stalls is shown for each miss policy and high water mark. Each is normalized such that Fetch-on-Write with a high water mark of 4 is 100%.

to 40% as one increases the write buffer's high water mark.

As the previous figures have demonstrated, the disadvantages of a large high water mark (write buffer and load service stalls) trade off against its advantage (reduced memory traffic due to coalescing hits). The performance impact of these opposing effects also depends on the overall frequency of loads and stores in the code. As Figure 6 shows, the overall effect of high water mark on performance is small for these applications, but not insignificant. In both applications, the total execution time decreases as high water mark increases until it reaches the value five for tsp and six for xlisp. At this point, the effects of buffer full and load service stalls begin to outweigh the benefits of coalescing, and execution time increases again. Overall, different choices of high water mark lead to a 6% performance range for tsp and a 15% performance range for xlisp.

5 Write Buffer Behavior, Compiler Optimizations and Application Characteristics

In this section we look in more detail at case studies that highlight compiler interaction with write behavior and write buffer design.

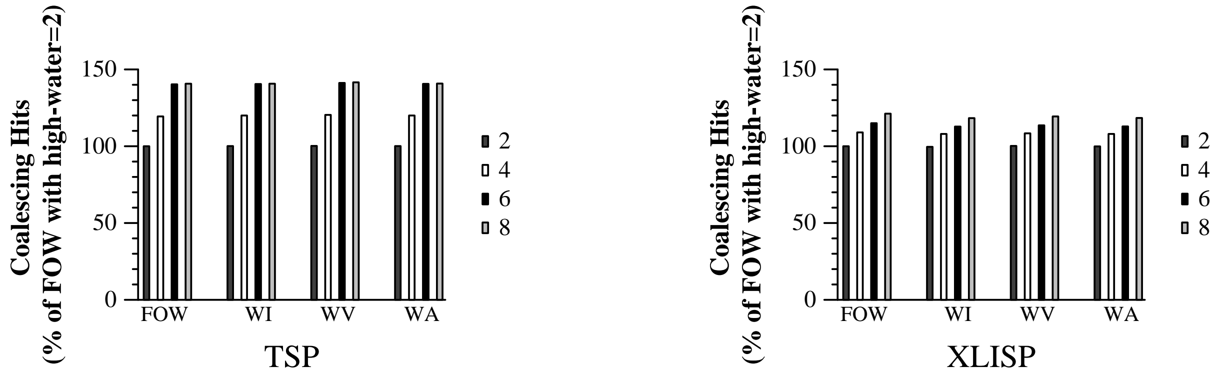


Figure 5: Frequency of coalescing write buffer hits as a function of high water mark. A count of coalescing hits is shown for each miss policy and high water mark. Each is normalized such that Fetch-on-Write with a high water mark of 4 is 100%.

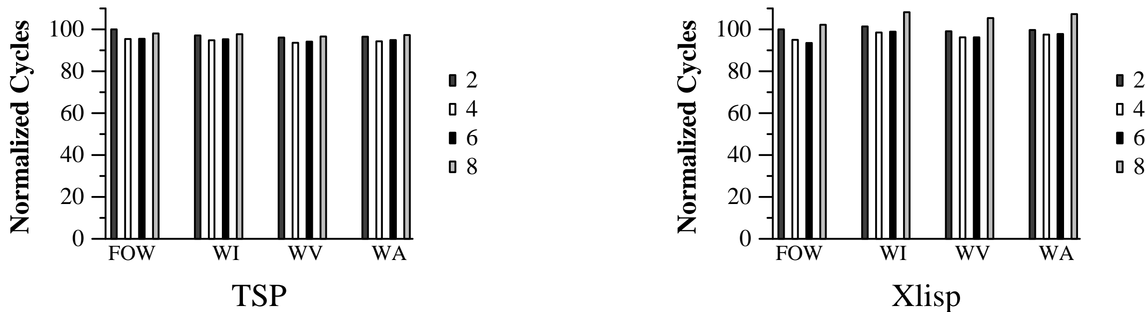


Figure 6: Effect of write miss policy and high water mark on overall application performance.

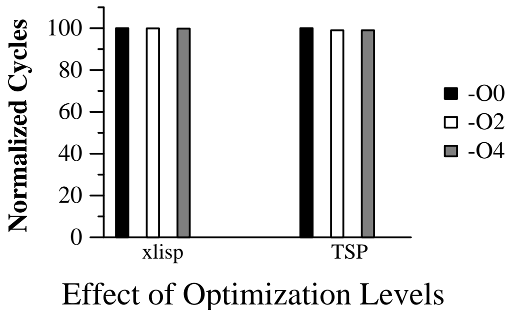
5.1 Compiler Optimization Levels

One initial goal of this research was to understand whether certain compiler optimizations would lessen the impact of write buffers, by reducing the memory operations that make write buffers important in the first place. A converse question also arose: would write buffers muffle the impact of compiler optimizations by making the memory operations (which might be optimized away by some compiler techniques) less costly in the first place? Figure 7 shows the normalized execution versus optimization level for xlisp and tsp. The left hand side graph shows the normalized performance when no write buffer is used. The graph on the right hand side shows performance with an 8 deep write buffer. Recall that, as in the previous section, performance with a write buffer (RHS) is about 3X better than without a write buffer (LHS).

As the graph on the left hand side shows, execution time in the no-write-buffer case is fairly flat. The three optimization levels show less than a 1% difference in performance. This is largely because the code is fairly resistant to optimization, and the bulk of the overhead is in memory latencies for this case. When we consider a system with a write buffer, the picture looks somewhat different. The graph on the right hand side of Figure 7 shows that TSP is still largely resistant to optimization. Xlisp, however, improves by roughly 5% as the optimization level is increased. Most of this improvement comes when moving from no optimization to the moderate optimization of -O2.

5.2 Procedure Calls and Returns

Based on original results such as those shown in Figure 7, we explored compiler/application/write buffer interactions in more detail. In particular, we instrumented our simulator to keep statistics on the program counter of the references that encountered buffer full stalls. From this data, we found that more than 50% of the time, buffer full stalls were associated with procedure entry or exit. That is, these stalls typ-



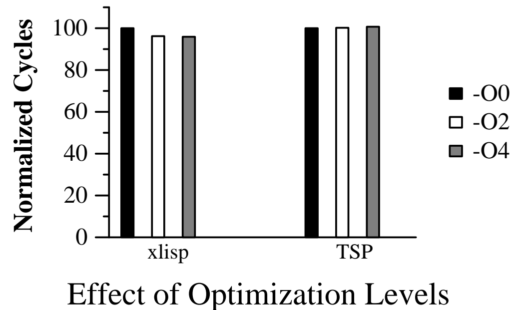


Figure 7: Effect of compiler optimization levels.

ically occurred within the stream of writes required to save out the caller-saved or callee-saved registers at a procedure call, or to pass in function arguments.

Similarly, load service stalls frequently occur at procedure entry and exit points as well. One reason for this is that if the parameter being passed was just stored in the calling procedure, a load service stall will likely occur as the location is read back in by the callee. If a function has relatively little computation, load service stalls may also occur when the return address is loaded. This is because the return address is stored upon entering the function and is loaded right before exiting the function. If there are few intervening instructions between entering and leaving the function (or if the high water mark is large) the return address may still be in the write buffer causing a load service stall. We found that between 13 and 18% of the load service stalls occurring for xlisp can be attributed to these reasons.

6 Conclusions

This work offers a preliminary view of the interactions between application characteristics, compiler characteristics, and write buffer behavior. We have found that the inclusion of a write buffer in a memory hierarchy tends to slightly magnify the importance of compiler optimizations and cache write policy. That is, application performance varies more widely with these parameters in systems with write buffers. In addition, we have found an important component of write buffer stalls (roughly 50% of buffer full stalls and roughly 15% of load service stalls) are related to procedure calls and returns in these applications. Based on these initial findings, we hope to continue this work by investigating the effect of procedure inlining techniques on write buffer performance in these codes, and by expanding our measurements to include a broader set of applications.

7 Acknowledgments

The bulk of this work was performed during the summer of 1996 when Kelly Shaw was at Princeton University. Her summer research stipend was funded through the NSF/CRA Distributed Mentoring Program. Margaret Martonosi is supported in part by an NSF Career Award (CCR-9502516).

References

[1] M. C. Carlisle and A. Rogers. Software Caching and Migration in Olden. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, July 1995.

[2] P. P. Chu and R. Gottipati. Write Buffer Design for On-chip Cache. In *International Conference on Computer Design*, pages 311–316, 1994.

[3] K. M. Dixit. New CPU Benchmark Suites from SPEC. In *Proc. COMPCON*, Spring 1992.

[4] N. P. Jouppi. Cache Write Policies and Performance. In *Proc. 20th Annual International Symposium on Computer Architecture*, May 1993.

[5] R. Karp. Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane. *Mathematics of Operations Research*, 2(3):209–224, Aug. 1977.

[6] K. Skadron and D. W. Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, Feb. 1997.

[7] J. E. Veenstra and R. J. Fowler. MINT Tutorial and User Manual. Technical Report Technical Report 452, Univ. of Rochester Computer Science Department, June 1993. Revised August 1994.