Characterization of Problem Stores

Allison L. Holloway and Gurindar S. Sohi
Department of Computer Sciences
University of Wisconsin at Madison
Email: {ahollowa, sohi}@cs.wisc.edu

Abstract— **This paper introduces the concept of problem stores: static stores whose dependent loads often miss in the cache. Accurately identifying problem stores allows the early determination of addresses likely to cause later misses, potentially allowing for the development of novel, proactive prefetching and memory hierarchy management schemes.**

We present a detailed empirical characterization of problem stores using the SPEC2000 CPU benchmarks. The data suggests several key observations about problem stores. First, we find that the number of important problem stores is typically quite small; the worst 100 problem stores write the values that will lead to about 90% of non-cold misses for a variety of cache configurations. We also find that problem stores only account for 1 in 8 dynamic stores, though they result in 9 of 10 misses. Additionally, the problem stores' dependent loads miss in the L2 cache a larger fraction of the time than loads not dependent on problem stores. We also observe the set of problem stores is stable across a variety of cache configurations. Finally, we found that the instruction distance from problem store to miss and problem store to evict is often greater than one million instructions, but the value is often needed within 100,000 instructions of the eviction.

I. Introduction

Ever-increasing memory latencies have caused computer architects to think about novel techniques to deal with memory latencies incurred on cache misses. A common characteristic of these techniques is *prefetching*, where the address that is likely to cause the cache miss is determined and submitted to the memory well before the corresponding load is encountered, allowing the latency of the memory operation to be overlapped with other computation. As relative memory latencies increase, computer architects will be called upon to invent even more techniques for proactive prefetching and memory hierarchy management.

A prefetching scheme has two aspects: *what* to prefetch, and *when* to prefetch. We are concerned with novel techniques to determine *what* to prefetch well in advance of the actual load that results in a miss. To put our proposal and previous techniques in context, consider the timing of events for a write-allocate cache in Figure 1. At some point in time during the execution of the program, A, a store instruction (at PC1) creates a value which it puts in address X. Later loads and stores may access address X. At time B, the line containing address X is replaced from the cache, resulting in a miss when the load (at PC2) accesses address X at time C.

Manuscript submitted: 17 Nov. 2004. Manuscript accepted: 16 Dec. 2004. Final manuscript received: 21 Dec. 2004.

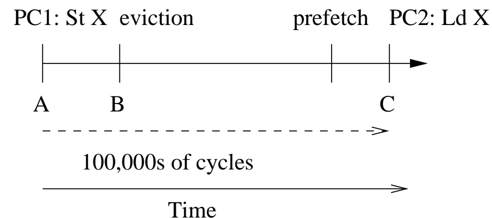

Fig. 1. Sequence of Events from Store to Load Miss.

Many prefetching schemes start with static loads that are likely to result in a cache miss (at PC2) and try to determine the address (X) that will be accessed by these loads. Prefetches to these addresses are then issued so that they can complete ahead of when the load is executed. The address calculation for these prefetches, for all but the most basic access patterns which can be accomplished in hardware, typically requires some software calculations. These calculations could either be carried out in the main program thread, as with software prefetching [7], or advanced loads (as in IA-64 [2]), or in a separate helper or prefetch thread [4], [8]. It is by now well established that a small number of static instructions cause a majority of the cache misses in user programs [1], [4], [9]. Prefetching schemes typically restrict their attention to this small set of static load instructions.

Looking at Figure 1 we see that a completely different approach to knowing the address (X) that will result in cache misses, well in advance of the cache miss, is to consider the store instruction (at PC1) which first writes into that address. If we knew when a store was performed, that the address to which the value is written will eventually be the object of a load miss, i.e., the store was a *problem store*, we could get the miss address at point A, far in advance of the actual load instruction, and with little effort since the store calculates the address anyway. Further, if the stores could be identified, the addresses could be marked and this information used for different cache management techniques. The challenge, however, is to identify and understand these problem stores, and distinguish them from other stores. Addressing this challenge is the subject of this paper.

Recent work has shown that static stores and loads have stable dependence relationships [3], [6]: thus, a dynamic instance of a static load only depends on the dynamic instances of a few static stores. This set, called the *store set* by Chrysos and Emer, typically has very few entries (often one) [3], [6].

Combining two different, independent ideas: (i) a few static

loads are responsible for a disproportionate number of cache misses, and (ii) the values accessed by a static load are produced by a few static stores, leads us to the main idea: *we can expect a few static store instructions to create the values that will eventually lead to load misses*. These static stores are the problem stores that we would like to distinguish from other stores.

We present a characterization of these problem stores. First, we show that there is, indeed, a small set of stores whose dependent loads cause most of the misses in the program. We then show that these problem stores make up a relatively small portion of all dynamic stores; their contribution to the total number of misses is much greater than their contribution to the total number of stores. Additionally, in L2 caches, the dependent loads of problem stores miss a higher percentage of the time than loads not dependent on problem stores. Finally, we explore distances from problem store to load miss, problem store to evict, and evict to load miss so that we can determine how far in advance we can identify a potential miss, how long the line stays in the cache, and how long we will have to manage the line until a load will need it.

II. Characterizing Problem Stores

A. Problem Store Identification

We define a *problem store* to be a static store whose dependent loads often miss.

We examine the miss behavior for the entire Spec2000 CPU benchmark suite (both floating point and integer) run from the beginning for 1 billion instructions with the train input set. We also study three commercial workloads: Apache, SpecJbb and OLTP. These are also run for one billion instructions, but start at a point where the caches are warm. Since we do not start from the beginning for the commercial benchmarks, our tracking does not include every store written by the program; however, we feel our results are still representative of the program behavior. We use a Simics-based SPARC Solaris full-system simulator [5], with our own cache model attached so we can model the events to our own level of detail. Simics feeds the cache one instruction at a time, stalling on cache misses, which are write-allocated. Simics's internal cache has been turned off.

We track all loads and stores by their word address. On a load hit or miss, we match the load with the static store that had last stored to that address and update its hit/ miss information, assigning a miss to the matching store. We then sort all static stores by the total number of load misses that were dependent on dynamic instances of the store. Cold misses and misses where we did not capture the writing store are not counted.

B. Characteristics of Problem Stores

Figure 2 shows the average (geometric) fraction of non-cold misses attributed to the 10 (black), 50 (gray) and 100 (white) worst problem stores for each class of benchmark for a 8K (left), 128K (left middle) and 1M (right middle) one-level cache and for a two-level cache with a 128K L1 and 1M L2 (right). All L1 caches are 2-way set associative, all L2 caches are 4-way set associative, and all caches have a 64 byte line size.

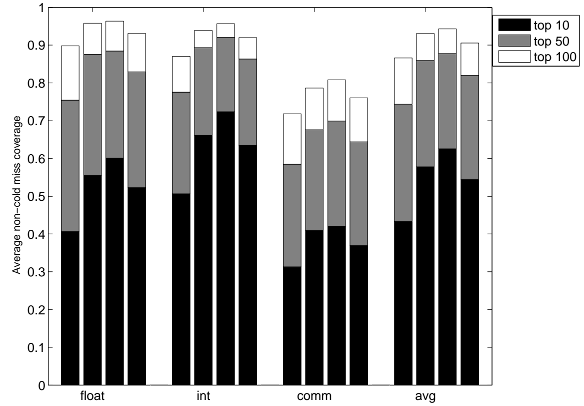

Fig. 2. Fraction of non-cold misses covered by the top 10, 50 and 100 worst problem stores for a 8K (left), 128K (left middle) and 1M (right middle) one-level cache and two-level cache with a 128K L1 and 1M L2 (right).

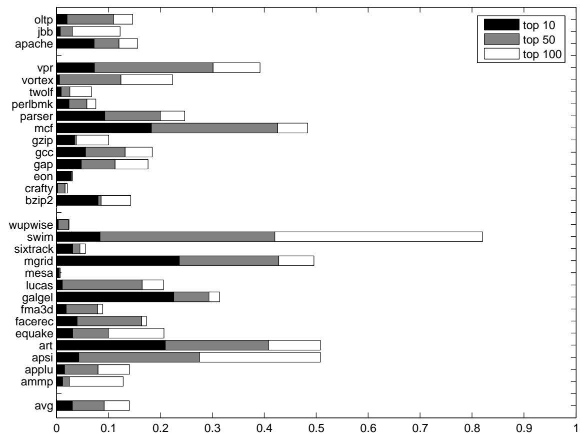

Fig. 3. Fraction of all dynamic stores caused by the top 10, 50 and 100 problem stores for a 128K one-level cache.

We call the percentage of all non-cold misses attributed to a set of problem stores the *miss coverage* or simply *coverage* of that set. As the figure shows, for a 128K cache, the top 100 problem stores have a coverage of roughly 93%, the top 50 have a coverage of 85%, and the top 10 are often enough to cover 58% of all non-cold misses. Overall, the 1M one-level cache has the highest fraction of load misses dependent on problem stores, but all the results are similar.

The commercial workloads tend not to have as high a coverage, though it is still relatively high at 70-80%; this could be the result of a larger working set, or the possibility of slightly skewed results from missed stores. Taken together, this data implies that, as expected, the set of problem stores which accounts for a significant portion of the non-cold cache misses is quite small.

Figure 3 shows the fraction of dynamic stores caused by dynamic instances of the top 10, 50, and 100 problem stores in a 128K cache. Comparing this figure with the previous

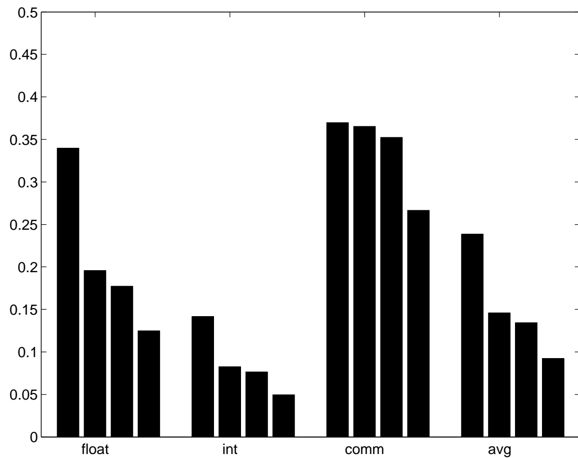

Fig. 4. Fraction of time the dependent loads of problem stores miss in the L2 cache for the top 10 (left), top 50 (middle) and top 100 (right) problem stores. The cache has a 128K L1 and a 1M L2. The rightmost bar is the L2 miss ratio for all loads for which we have seen their writing stores.

figure suggests that problem stores contribute to a much larger fraction of misses than of references. The top 100 problem stores account for one in eight store references, yet typically account for nine out of ten misses. The numbers are even more dramatic for top 10 (and top 50) problem stores: they account for about 3% of all stores, but cause about 60% of all non-cold load misses.

C. Cache Hit/Miss Statistics for Problem Stores

Figure 4 shows the average (geometric) miss ratios of the L2 cache. We define miss ratio to be the number of load misses divided by the total number of loads, for a particular set of loads. The three leftmost bars for each class are the miss ratios in the L2 for the dependent loads of the top 10 (left), 50 (middle) and 100 (right) problem stores for a two-level cache with a 128K L1 and 1M L2; the rightmost bar is the L2 miss ratio for all loads for which we have seen the writing store.

In general, the worse the problem store, the higher the load miss percentage. Additionally, for most of the benchmarks, the miss ratio for the top 10 problem stores, 25%, is significantly worse than the total miss ratio, 9%, and even the miss ratios for the loads of the top 50 and 100 problem stores.

D. Stability of the Set of Problem Stores

The previous data suggests that very few problem stores account for most of the cache misses. However, the results do not show if there is any overlap between the set of problem stores for different cache configurations. If the set of problem stores is stable across a variety of cache configurations, then this set could be detected for one cache configuration, and used as a proxy for the problem store set for another configuration.

We measure the stability of the problem store set between a collection of different cache configurations by counting the number of problem stores all the configurations have in common. Figure 5 shows the stability of the top 10, top 50 and top 100 problem stores between an 8K, 128K and 1M

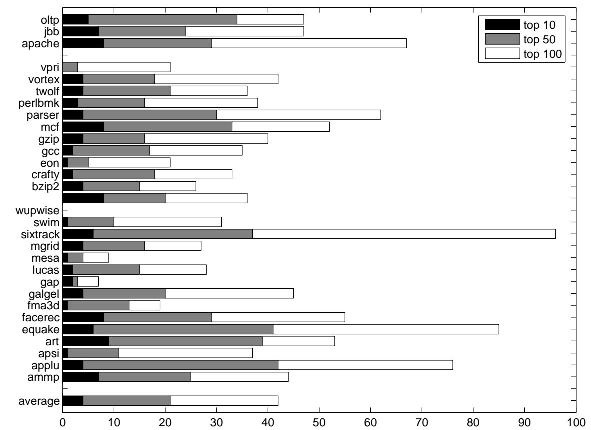

Fig. 5. Number of problem stores in the intersection of top 10, 50 and 100 problem store sets for an 8K, 128K and 1M 2-way set associative cache with 64 byte lines.

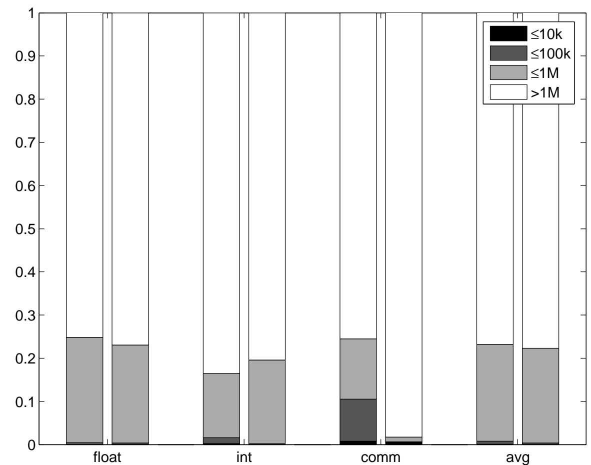

Fig. 6. Breakdown of distance in instructions between a miss and the dynamic problem store that wrote its address in a 128K one-level cache (left) and two-level cache with a 128K L1 and 1M L2 (right).

one-level cache (all 2-way set associative with 64 byte lines) for each benchmark. If the sets of problem stores were exactly the same, the line for top 10 would be at 10, top 50 at 50 and top 100 at 100. Instead, overall, the top 10 share 4, top 50 share 21, and top 100 share roughly 42. Some of the variance can arise because cache configurations can slightly change the relative importance (and hit/ miss) characteristics of stores, thus making the static stores appear in a different order in the rankings. The above data suggests that if we can capture the 100 worst problem stores for an 8K cache, on average we would be able to also capture roughly 42 of the worst for both a 128K and 1M cache.

E. Distances between store, evict and miss

Figure 6 presents the breakdown of distance between the dynamic problem store instance and the resulting load miss. The bars are broken down by distance: less than 10,000, 100,000, and 1,000,000 instructions and greater than one million instructions; the larger the bar, the more often that

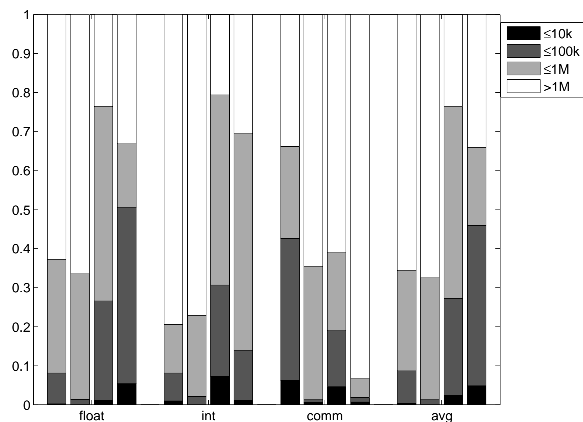

Fig. 7. Breakdown of distance in instructions between store and evict (left two) and distance in instructions between evict and load miss (right two). Each set has results for a 128K one-level cache (left) and 128K L1 with 1M L2 (right).

range of instruction distances occurred.

The figure shows that 76% of misses attributed to problem stores occur at least one million instructions after the problem store for a 128K cache, and 77% of the time for the L2 configuration. Additionally, almost every miss occurred greater than 100,000 instructions after the problem store wrote its data. This data suggests that if problem stores are identified correctly, the address of a future miss can be determined trivially, long in advance of the miss, perhaps allowing for proactive prefetching schemes with the ability to tolerate latencies of tens of thousands of cycles or larger.

We do not necessarily have to use knowledge of a future load miss as soon as the value is stored. If we know, at the store, that an instruction will load-miss on the address, we can mark it and wait to manage it in the higher-level caches or when the line is evicted from the cache. Figure 7 shows two sets of results: the distance (in instructions) from store to evict (left set) and from evict to load miss (right set). Each of these sets has two bars: one for a 128K cache alone (left) and one for 128K L1 with 1M L2 (right). The address ranges are the same as in the previous section.

There are more than a million instructions between the store and evict 55% of the time, and from 100,000 to 1,000,000 instructions another 45% of the time, but evict to miss results are shifted down one category: 70% of the time there are 10,000 to 1,000,000 instructions between the two; there are more than one million instructions between the two only 30% of the time. These results make the problem more tractable: if the address is marked as coming from a problem store, 30-45% of the time we only need to manage the line for another 100,000 instructions, rather than a million or more.

III. Conclusion

This paper introduced the concept of problem stores, stores that write into a memory location an access to which will later result in a load miss. A characterization of problem stores was presented. The characterization suggests that a small set of problem stores results in a disproportionate number of cache misses. The top 100 problem stores account for 93% of all non-cold cache misses. The contribution of problem stores to the number of cache misses was far greater than their contribution to the total number of stores, and the loads that read from the stores have high miss ratios.We also observed that the set of the top problem stores was reasonably stable over a variety of cache configurations. Finally, we looked at the distance in instructions from store to load miss, store to evict and evict to load miss. The store distance statistics show that there are millions of instructions from store to miss and almost that many from store to evict; however, there are significantly less from evict to miss, suggesting that the problem should be more tractable with good memory hierarchy management.

Some of the empirical observations made from the data in this paper could, in hindsight, be made independently, and non-empirically, by putting together the empirical observations made in two independent bodies of prior work: (i) a few static loads account for most cache misses [1], [4], [9], and (ii) most loads depend upon a few (typically one) static stores [3], [6].

Software and hardware identification of the set of problem stores opens up a plethora of possibilities for novel software prefetching and memory hierarchy management techniques. Having introduced the concept of problem stores in this paper, we can also expect a lot of future work in software techniques to exploit this concept.

Acknowledgments

This work was supported in part by National Science Foundation grants EIA-0071924 and CCR-0311572 and the University of Wisconsin Graduate School. Allison Holloway was supported by a National Science Foundation fellowship.

References

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 139–152, 1993.

[2] B.-C. Cheng, D. A. Connors, and W. mei W. Hwu. Compiler-directed early load–address generation. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 138–147, 1998.

[3] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, 1998.

[4] J. D. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: long–range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, 2001.

[5] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallber, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[6] A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Symposium on Computer Architecture(ISCA)*, pages 181–193, June 1997.

[7] T. Mowry and A. Gupta. Tolerating latency through software–controlled prefetching in shared–memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[8] A. Roth and G. S. Sohi. Speculative data–driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 37–48, January 2001.

[9] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, June 2001.