Software Methods to Increase Data Cache Performance

Philip A. Marshall, *University of Alberta*

I. Introduction

Cache performance is critical to the overall performance of modern CPUs. In most processors, cycle time is almost entirely determined by the cache hit time. This places practical limits on the complexity of cache controllers. Because of the growing gap between CPU performance and memory access times, the cost of a cache miss in CPU cycles is growing steadily [1]. As a result, it becomes critical to use caches effectively. More complex replacement schemes have been proposed to predict which data is most likely to be useful in the cache [8]. In addition, complex hardware prefetch schemes have been developed to predict which data is likely to be needed in the future [3,6,7]. However, such modifications are likely to increase the hit time of the cache, and hence the cycle time of the processor.

Of particular interest are software schemes to improve cache performance [2,4,5], as they require little or no modifications to the instruction set architecture. In particular, they do not greatly increase the complexity of the cache controller. They also hold the potential to be mined by compilers, having essentially no negative effect on the cost of software development. This paper will examine several such schemes and present their relative costs and benefits. Loop fusion increases the temporal locality of accesses by combining multiple loops that access the same elements. Array merging increases spatial locality by combining multiple arrays whose elements are always accessed together. Cache prefetch reduces cache misses by ensuring that the data is in the cache in advance of when it is needed. The effect of these techniques on example programs is examined.

II. Cache Assumptions

Throughout this paper, unless otherwise specified, the memory hierarchy will be a 2-level cache system with a line size of 4 words. A level 1 miss will be assumed to take 10 cycles, while a full cache miss will take 200 cycles total. Instruction cache misses are ignored, as are conflict data misses. We assume that write buffers can be used to hide the cost of write misses, and focus only on read misses. Cache miss calculations were done by hand, due to problems with the SimpleScalar compiler that prevented simulation from being used.

III. Loop Fusion

Loop fusion is a technique to improve the temporal locality of data. It is useful when the same data is being fetched multiple times in different loops [1,2,4]. Before loop fusion, and with sufficiently large data sets, the useful data does not remain in the cache, but is replaced before it is accessed again. By combining these loops together, the data only needs to be fetched once, generating fewer cache misses. For example, the program vector.c in Appendix A calculates the sum of vectors. Vectors a[] and b[] are added to yield sum s1[], and vectors a[] and c[] are added to yield sum s2[]. As written, each element of a[] must be read twice, once in each loop. If the cache is not large enough to hold the entire vectors of a[] and b[], the second access to each element in a[] will result in a cache miss. Program vector_fused.c combines the two loops into one, resulting in the reuse of the cached value of each element in a[].

It is possible that the shared accesses will only partially overlap. Program vector2.c adds two different partially overlapping segments of vectors a[] and b[] into sums s1[] and s2[]. Splitting each of the two loops into their overlapping and non-overlapping parts allows us to fuse the overlapping loop into one loop that combines accesses to the vectors a[] and b[]. This is seen in vector2_fused.c.

The program vector.c generates a cache miss on every fourth load of a[] and b[] in the first loop, and a cache miss on every fourth load of a[] and c[] in the second loop. With 40 million total loads, this generates 10 million cache misses. By simply fusing the loops, we can decrease the total number of loads to 30 million, generating only 7.5 million misses. In vector2.c, we see 4 million misses in the first loop and 4.5 million misses in the second loop. By fusing the loops as in vector2_fused.c, we get 500K misses in the first loop, 3.5 million misses where we can combine the overlapping regions, and 1 million misses in the final loop for 5 million misses total.

Besides the reducing the loop overhead by decreasing the number of loops, loop fusion allows us to improve cache performance. In general, this technique may be exploited by the compiler with varying degrees of aggressiveness depending on the potential benefit.

IV. Array Merging

Array merging increases spatial locality by combining data elements that are likely to be accessed together [1]. Accesses to consecutive memory addresses already benefit from spatial locality inherent in the fact that cache lines are larger than a single word of data. Therefore, array merging only provides an additional benefit for accesses that are farther apart than the cache line size. This may include random accesses, or accesses at regular intervals larger than the line size.

Program vector3.c creates a vector s1[] consisting of the sum of every 10,000th element of the vectors a[] and b[]. In program vector3_merged.c, the arrays a[] and b[] are merged into one array where each element is a struct with elements a and b. Therefore, the corresponding a and b elements are always fetched into the cache together, resulting in an improved hit rate. In vector3.c, each of the 20,000 loads will miss the cache, while in vector3_merged.c only every second load will miss since each of the corresponding elements of a[] and b[] are loaded together.

Array merging may have limited effectiveness in practice. For example, given several vectors, it is not a useful technique if we would like the ability to add elements from any two arbitrary vectors where we do not know at compile time which vectors we will use. Array merging is likely limited to cases where the merged arrays will inherently be accessed together, such as key-value pairs. In addition, array merging is likely dangerous for compilers to exploit. Changing the way structures are represented in memory could cause unexpected side effects, especially with shared memory or pointer arithmetic. As a result, this is likely a technique that must be explicitly programmed.

V. Cache Prefetch

Cache prefetch can be an effective way of hiding memory latencies [6]. By issuing requests far enough in advance for data we know will be accessed, we can ensure that it is in the cache when the load occurs, resulting in a cache hit. Two types of prefetches are possible, depending on the instruction set architecture. One is an advance load (as in the IA-64 architecture [1]) that requests in advance that data be loaded into a register. Then, before using the data in the register, a check instruction is executed that ensures that the data in the register is valid before proceeding. In this paper, we will consider the second type: non-binding prefetches. In this case, the prefetch instruction simply loads the data into the cache [5]. Loads are issued as normal. If the prefetch was successful, we get a cache hit. Otherwise we must wait for the data to finish being fetched from memory before continuing. We will assume that prefetching from memory that would cause a fault simply translates the prefetches into NOPs and that no fault occurs unless we try to issue a load to an invalid address. If this assumption does not hold, it is trivial to split the loops such that we do not issue prefetches past the end of arrays.

Consider the case of program vector_prefetch.c. Assuming 4 cycles for each loop iteration (two loads, an add, and a store, ignoring loop overhead), prefetching data 200 cycles ahead (to allow for an entire cache miss) requires requesting it 50 iterations in advance. The first 100 misses of each loop in vector_prefetch.c cannot be avoided, but we decrease the number of misses to 200 total. By combining prefetch with loop fusion (as in vector_fused_prefetch.c), we can further decrease the total number of misses to 150. Note that in vector_prefetch.c we generate 40 million dynamic prefetch instructions, while in vector_fused_prefetch.c we generate 30 million. The cost of issuing these instructions can be hidden if the processor can issue them in parallel with other instructions in the loop. Even if they cannot be hidden, they represent only 1 cycle loss, rather than 200 cycles for a full cache miss. The cost of the code size increase by adding the prefetch instructions must be considered, but is likely negligible in a microprocessor system.

Likewise, vector2_prefetch.c avoids the cost of all but the first 50 cache misses for each of a[] and b[] in each loop, generating 200 total misses. By fusing the overlapping parts of the loops (vector2_fused_prefetch.c), we can decrease the total number of misses to 100.

Some processors incorporate hardware prefetch units that automatically fetch the next few words at consecutive addresses on a cache miss. This prefetch may or may not be subject to a prediction as to whether these words will be useful. These help cache miss rates in the case of sequential access to consecutive addresses, but not in the case of non-consecutive access patterns. More complex prefetch predictors have been proposed [3,6,7], but at this point in time the hardware is likely too expensive for the value it provides. If the software can compute at runtime which memory is to be accessed, explicit prefetch instructions are likely a better solution.

The program linkedlist.c is an example of prefetching where elements may not be contiguous. A linked list is traversed and its elements copied into an array. Even with a sequential hardware prefetch mechanism, in the worst case we get 10 million load misses because the list is fragmented and has no spatial locality. If we assume 2 instructions per iteration (1 load and 1 store, ignoring overhead), we can prefetch 100 elements ahead in the list (linkedlist_prefetch.c). In this case, we need only miss on at most the first 100 elements.

While the cases above prefetch only data that we know we will use, we may also wish to speculatively prefetch data. The program binarytree.c outlines a binary tree search routine. Speculatively fetching N levels ahead in a binary tree would result in loading 2^{N-1} elements and using at most N of them. The never-to-be-used data would likely push useful data out of the cache, harming cache performance. However, if the search function is expensive to compute (for example, a complicated cost function), we may be able to prefetch the next two elements while we decide which one is needed. The program binarytree_prefetch.c gives an example of such a case.

With aggressive prefetching, we must consider the size of the cache [5]. Assuming the prefetch instruction loads data into the L1 cache, we are filling the L1 cache with data that we will not need for 200 or more cycles into the future. In the case of speculative prefetch, we may never use the data at all. It may be worthwhile to consider two prefetch instructions: one that prefetches data into the L2 cache, and one that prefetches data into the L1 cache. L2 prefetch instructions can be issued 200 cycles in advance, while L1 prefetch instructions for the same data need only be issued 10 iterations in advance. As the cache miss time in cycles continues to increase, data will need to be requested even further in advance of the actual load, requiring even more cache space.

Prefetching data into the cache can effectively hide the access latency as long as we're able to predict which data we need. In some cases, compilers may be able to determine which data to prefetch, leaving the programmer free to focus on other issues. In other cases, the programmer may explicitly program prefetch instructions, possibly even speculatively requesting data.

VI. Summary of Results

Table 1 shows the data cache miss rates for the programs previously discussed, as well as the number of additional prefetch instructions that must be executed. Loop fusion and array merging give modest results, while prefetching gives the most spectacular gains. However, the cost of executing, in some cases, several million prefetch instructions must be carefully weighed to determine whether there is a net gain. In some cases, multiple techniques can be combined to get better performance gains than any of the techniques achieve by themselves.

VII. Conclusion

With cache performance being so critical to processor performance, it is worth exploring methods to improve cache performance. Software methods can provide ways to do exactly that, with little hardware cost of implementation. By considering the way caches exploit locality, code can be arranged to increase spatial locality using array merging, or increase temporal locality using loop fusion. Both methods can be moderately effective under certain conditions. Prefetching can be extremely effective in the case of predictable memory access and a sufficiently large cache.

Table 1: Data Cache Misses and Dynamic Prefetch Instructions for Example code fragments

Program	***Data Cache Misses***	***Dynamic Prefetch Instructions***
vector.c	10,000,000	0
vector_fused.c	7,000,000	0
vector_prefetch.c	200	40,000,000
vector_fused_prefetch.c	100	30,000,000
vector2.c	9,500,000	0
vector2_fused.c	5,000,000	0
vector2_prefetch.c	200	34,000,000
vector2_fused_prefetch.c	100	20,000,000
vector3.c	20,000	0
vector3_merged.c	10,000	0
vector3_prefetch.c	100	20,000
vector3_merged_prefetch.c	50	10,000
linkedlist.c	10,000,000	0
linkedlist_prefetch.c	100	10,000,000

REFERENCES

[1] Hennessy, John L.; Patterson, David A., “Computer Architecture: A Quantitative Approach”, *Morgan Kaufmann, 2003*

[2] Ng, J.; Dattatraya Kulkarni; Li, W.; Cox, R.; Bobholz, S., "Inter-procedural loop fusion, array contraction and rotation," *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on* , vol., no.pp. 114- 124, 27 Sept.-1 Oct. 2003

[3] Oliver, R.L.; Teller, P.J., "Dynamic and adaptive cache prefetch policies," *Performance, Computing, and Communications Conference, 2000. IPCCC '00. Conference Proceeding of the IEEE International* , vol., no.pp.509-515, Feb 2000

[4] Manjikian, N., "Combining loop fusion with prefetching on shared-memory multiprocessors," *Parallel Processing, 1997., Proceedings of the 1997 International Conference on* , vol., no.pp.78-82, 11-15 Aug 1997

[5] Xiaotong Zhuang; Hsien-Hsin S. Lee, "Reducing Cache Pollution via Dynamic Data Prefetch Filtering," *Computers, IEEE Transactions on* , vol.56, no.1pp.18-31, Jan. 2007

[6] Vander Wiel, S.P.; Lilja, D.J., "A compiler-assisted data prefetch controller," *Computer Design, 1999. (ICCD '99) International Conference on* , vol., no.pp.372-377, 1999

[7] Johnson, T.L.; Merten, M.C.; Hwu, W.W., "Run-time spatial locality detection and optimization," *Microarchitecture, 1997. Proceedings. Thirtieth Annual IEEE/ACM International Symposium on* , vol., no.pp.57-64, 1-3 Dec 1997

[8] Chi, C.-H.; Dietz, H., "Improving cache performance by selective cache bypass," *System Sciences, 1989. Vol.I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on* , vol.1, no.pp.277-285 vol.1, 3-6 Jan 1989