Synchronization-aware Prefetching

Dan Le
danle@umich.edu
Junjie Wu
wujj@umich.edu
Hongyi Xin
xhongyi@umich.edu

ABSTRACT
Data prefetching is an important part in increasing the performance of the majority of programs written. Most of the designs for data prefetchers today try to find patterns in the data access. Other prefetchers attempt to correlate addresses, so that the access to one address is indicative of another address in the near future. Multi-threaded programs use locks or barriers to synchronize accesses to shared data. The data that is shared between threads of a program tend not to move much in memory, due to the desire to keep as much of the memory processor specific to avoid coherence misses and costs. The prefetcher that we propose in this paper takes advantage of how those programs use locks and the nature of shared data in general.

In this project, we examine the effectiveness of using synchronization primitives as prefetch hints to improve performance in scientific workloads.

1. INTRODUCTION
Current prefetchers relate the prefetching data with the current memory address. This method works pretty well for instruction prefetching as instructions are continuous for a vast majority of the time and prefetchers can work with the branch predictor to prefetch instructions across a taken branch. However, this doesn’t work with data prefetching. The problem of this is that data doesn’t usually have a nice pattern throughout the program and they are also affected by the control flow (i.e. branches). In addition, the possible addresses touched by data read/writes are too large to be recorded in a table. There has already been previous work done on various prefetchers that correlate the program counter with data, or ones that exploit patterns in the data access.

In our study, we explore the possibility of using synchronization primitives as prefetch hints to both increase the accuracy and reduce the index space in traditional data prefetchers. The paper is organized as follows. Section 2 discusses previous works on data prefetch, and their advantages and weakness. Section 3 describes the detailed design of our prefetcher. Section 4 describes the simulation environment and experimental setups. We present our results and conclusion in section 5 and 6, respectively.

2. BACKGROUND
Historically, there has been a lot of designs to help improve the performance of cache. One of the most effective techniques is to prefetch data into the cache in advance of executing the real load or store instruction. Issuing data load from memory in advance of the actual use of data would increase the parallelism of memory access. Because of the long latency of memory accesses, fetching data in advance and fully utilising the memory bandwidth is critical for performance, especially if the addresses and the data pattern are well constructed and predictable. Programs that share this attribute like scientific applications could gain a lot of performance speedup if the prefetcher can fetch the data in advance with 100 percent accuracy. However, assuring 100 percent accuracy in data prefetching turns out to be pretty hard as there are no guarantees the application will access the same memory even if its the same code.

Several mechanisms have been studied to attack this problem and improve the data prefetching accuracy without a huge performance impact, including software-based prefetch schemes and hardware-based prefetch schemes.

Software prefetching is one method that can drastically improve performance [4], but to efficiently prefetch data, the compiler must be able to reliably predict memory access patterns, which is hard to predict as the compiler has no idea how the memory access pattern would be at run time. This approach also limits the portability of the program as different hardware configuration may have different memory accesses thus an efficient software prefetching scheme on one machine may behave poorly on another machine.

Hardware prefetching schemes, on the other hand, are tailored to specific hardware layouts and are able to make prefetch decisions at run-time but have lack of information. Thus figuring out where to prefetch is the most basic question in hardware prefetching. The simplest way of hardware data prefetching is sequential prefetching [3], where upon a cache miss of cache line A, the memory will fetch the missing cache line A together with another cache line B that is K lines ahead of A (B = A + K). Notice that, although the hardware is simple and fast, the accuracy is low and thus may cause potential memory communication contention while polluting the cache by prefetching useless data. Stride prefetching [1] improves the accuracy by
allowing different distance $K$, i.e stride distance by their definition, for different memory operations in the applications. However, this mechanism only works well in the case that the load it will prefetch will be executed for many times and the stride distance between executions is fairly constant. Notice that for most of the loads in program, they are executed only once or limited times so a stride buffer can mostly only help with data within loops. Another mechanism called block prefetching, or more generally correlated prefetching, [5, 6] stores subsequent load information with current load and upon a load it can do chain prefetching of the subsequent memory loads. However, storing correlating memory operation information is extremely expensive and the performance gain may not pay back the storage overhead required.

Thus, what we need is a prefetch mechanism that incurs little storage overhead but with a high accuracy and lock may be considered as a qualified hit in correlated prefetch rather than normal memory operation.

3. DESIGN

In the end, we tested two designs.

3.1 Theoretical Design

The idea for the prefetcher that we designed is to record what addresses were accessed in between an atomic store to a memory location and the subsequent store to the same memory location. The next time that the memory location is atomically stored to, the prefetcher will activate and prefetch all blocks that were accessed during that section before. At the same time, it will start remembering what addresses were accessed between the current atomic store and when that memory location gets stored to. We decided to correlate this list with one of three values: the program counter of the atomic store, the address that the atomic store used, and the XOR of the preceding two values. The three values capture the cases respectively where a given lock address would only be used at one program counter, where a given lock address was used at multiple program counters but protected the same data, and where the same lock address may be used at different program counters and protected different data.

Given that most locks exist for protecting the same regions of memory over the lifetime of a program, we expect our prefetcher to prefetch mostly the same pieces of data each time the lock is encountered.

To address the problem of atomic stores being used merely for data manipulation and not for a lock, we implemented a cap on how many addresses could be prefetched. In an actual implementation, it should be possible to turn the prefetching feature on or off for a given atomic store, to capture only those stores that actually exhibit the behavior that we desire.

3.2 Realistic Design

The assumption of unlimited recording entries is un-
realistic in hardware design. Therefore, we designed a cache-like structure as our prefetcher. The prefetcher is fully-associative, with 32/64/128 entries indexed by the xor results of PC and memory addresses. We use LRU replacement policy for these entries. Within each entry, prefetcher records 4/8/16/32/64 subsequent normal memory accesses.

Each time there is a RMW operation, we check the prefetcher. If the index hits in the prefetcher, we check all recorded memory accesses with cache. Cache will filter out addresses that are already in its tag array, and we prefetch all other addresses into the prefetcher. We use non-binding prefetch due to the mediocre accuracy we found in theoretical analysis. Cache hit addresses are not visible to prefetchers, except for RMW accesses, which are used to convey index information to the prefetcher.

4. METHODOLOGY

We evaluated our design using Simics, which is a virtual platform executes unmodified operating systems and applications. The simulated machine has 16 cores, runs an unmodified Solaris 10, on Sparc v9 ISA. The simulator round-robins between 16 cores in serial and run 1 instruction from each core each time. The instructions and memory accesses were captured by Simics default memory tracer and fed into our prefetch simulator. There is no timing model in this setup and every instruction execution and memory access happen atomically.

We used scientific workloads provided by PARSEC[2] benchmark to evaluate our prefetcher design. We chose simlarge as the input set for all benchmarks in PARSEC. Due to the long simulation time and large memory usage to track all addresses, we only run 16 million instructions in every 500-million instruction interval.

We modeled a set-associative cache with LRU replacement policy for each CPU core. The cache is private to each core and we used MSI protocol among all caches. Since we dont have a timing model, all coherence transactions happen atomically as well. We warmed up our cache for 1 million cycles before recording statistics. We modeled two caches separately, a 64KB 2-way set-associative cache (typical size of L1) and a 1MB 8-way set-associative cache (typical size of L2).

5. RESULTS

We present our results in three sections. Section 5.1 discusses the characteristics of PARSEC benchmarks, focusing on how closely synchronization primitives are correlated with data access. Section 5.2 discusses different schemes we have explored as prefetcher indexing hints, and their pros and cons in terms of accuracy. Section 5.3 discusses the index space and its implications. Section 5.4 shows the effect of prefetch depth on accuracy and relative coverage.

5.1 Characterization

Figure 1 (Data, sheet3, pic) shows the best-scenario accuracy for PARSEC benchmarks, with infinite prefetch depth. Here we assume all addresses are instantly prefetched whenever an index matches. All normal loads and stores are recorded until a lock release or a new atomic store occurs. We will refer to this scheme as infinite prefetch depth in later discussion.

Stopping prefetch upon detection of a lock release filters out lots of prefetch misses in low accuracy bins. The same effect is observed when we reduce the prefetch depth. Please refer to 5.4 for detailed discussion.

From [2], blackscholes is barrier-only; bodytrack, canneal and streamcluster have both locks and barriers. All other benchmarks are lock only. Figure 1 suggests
different synchronization primitives do not have huge impact on prefetch opportunities.

5.2 Indexing Scheme

Figure 1 also lists the accuracy of different indexing scheme, 1) virtual address of atomic store 2) virtual address of PC which issues read-modify-write 3) XOR result of the previous two addresses.

Scheme 1 is effective because lock address indicates the location of shared data it protects. It will mispredict when a lock is used in different parts of the program to execute different tasks, or the lock is too coarse-grained to be specific hint for certain memory address.

Scheme 2 is effective because PC indicates the execution of program, which is a useful hint observed by code-based prefetchers [6]. It will mispredict when the instruction locks different locks and access different data the locks protect.

In a multi-threaded program, a lock statement can lock different locks, whereas a lock can be used in different segments of a program. Scheme 3 takes advantage of both patterns, and subsumes previous two schemes.

Therefore, scheme 3 gives best accuracy in all benchmarks.

5.3 Indexing Space

The total number of indices ranges from thousands to millions, even for our limited trace length. We attempted a full-run of canneal and it quickly exhausted all 16G host memory. Therefore, keeping track of all indices on-chip is impossible. Meanwhile, from the total number of prefetch hits (not shown in graph), average hits on each index is quite small, which suggests we can cache limited number of indices and replace inactive lines. We suggest a cache-like structure, indexed by read-modify-write PC xor address, with high set-associativity to combat the index explosion. A good replacement policy to quickly filter out non-promising indices is required to achieve meaningful performance.

5.4 Effect of Prefetch Depth

Figure 2-4 show three kinds of behaviors when we change the prefetch depth. We normalized the "coverage" with respect to unlimited depth prefetch discussed before. The right most data point represents coverage and accuracy of unlimited depth prefetch.

In the first group, represented by canneal (Figure 2), accuracy is not very sensitive to the prefetch depth, and best coverage can be achieved by recording small number of addresses for each index. This behavior is very friendly to synchronization-aware prefetch because good performance can be achieved with low hardware cost.

In the second group, represented by Facesim (Figure 3), its impossible to achieve the best coverage. However, reasonable coverage and accuracy is possible for limited prefetch depth.

In the third group, represented by Raytrace (Figure 4), coverage is very low for a limited prefetch depth. This behavior prevents reasonable prefetch performance with limited hardware.
6. CONCLUSION

From our results, synchronization-aware prefetch is not promising enough to achieve good performance on chip. The index space is greatly reduced, but it is still too large to fit on-chip. The accuracy is not high enough, even with the assumption of instant prefetch. The coverage is hurt more by limiting the prefetch depth, which is inevitable when designing the hardware. Without a timing model, we cannot draw a firm conclusion about its performance influence, but we expect the benefit of synchronization-aware prefetch to be worse than most of other prefetch methods.

7. CONTRIBUTIONS

Hongyi coded the credited based prefetcher (Section 5.5).

Dan coded the theoretical analysis experiment (Section 5.1-5.4).

Junjie generated and parsed Simics trace as the backend of Dan/Hongyi’s code. Junjie also provided simple multicore cache for Hongyi.

Everyone ran experiments and collected results.

Wasted but non-trivial efforts:

Hongyi tried to bring up SPLASH-2 in Pin. Pin was dropped because Simics is more powerful in identifying atomic instructions/addresses. Dan tried to bring up m5. M5 was dropped because of its complexity and we don’t really need a timing model for this study.

8. REFERENCES


