Data Prefetching Mechanism by Exploiting Global and Local Access Patterns

Ahmad Sharif
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: ahmad@gatech.edu

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: leehs@gatech.edu

***Abstract*—This paper presents a new hardware prefetcher based on the idea of the Global History Buffer proposed in [4]. We extend this idea to Local History Buffers, which keep the memory access information for selective program counters. These buffers can then be queried on cache accesses to predict future memory accesses and enable data prefetching. Our trace-driven simulations show that by using approximately a 4KByte (32 Kbits) storage budget, an average performance improvement of 20% (geomean) can be obtained for SPEC benchmark suite on an ideal out-of-order processor.**

I. Introduction

Single-thread performance can be severely impaired by long-latency cache misses. These cache misses can cause the re-order buffer to fill up and allocation to stall for tens or even hundreds of cycles, blocking forward progress. To address this issue, a significant amount of transistor asset in recent microprocessors is either dedicated to reduce the number of such misses by enlarging the cache capacity and/or hide the latency via an aggressive prefetcher. In this paper, we investigate the design and implementation of a hardware prefetcher under the storage budget of 4KB with allowable logic complexity. Our main contributions are as follows:

1) We identify and classify commonly occuring access patterns of memory instructions.
2) We propose a 4KB hardware prefetcher which demonstrates high effectiveness in data prefetching.

The rest of the paper is organized as follows. Section 2 describes some of the common access patterns in the SPEC benchmark suite. Section 3 describes the design of our data prefetching scheme. Section 4 discusses of the simulation infrastructure and our simulation methodology. Section 5 presents the results while Section 5 concludes.

II. Motivation & Observed Access Patterns

To gain an understanding of the cache behavior of modern workloads and glean useful information for effective data prefetching, we used traces generated from the SPEC 2006 benchmark suite [5]. For each run, we skipped the first 40 billion instructions and used the next 100 million instructions to analyze the memory access characteristics of the benchmarks.[1] As part of an initial study, we investigated the limit of a prefetching scheme by running simulations with a zero-latency L2 and/or DRAM memory.[2] This gives us an approximate upper-limit on what an ideal prefetcher could achieve. Our simulations were run for three different configurations provided by the DPC-1 infrastructure and recapped below:

1) Configuration 1 simulated an ideal 4-issue, out-of-order processor with no branch hazards. The L2 cache was 2 MB and bandwidth to the caches and memory was unlimited.
2) Configuration 2 simulated the same processor as Configuration 1, but with limited L2 bandwidth of 1 request per cycle and limited DRAM memory bandwidth of one request every 10 cycles.
3) Configuration 3 simulated the same processor as Configuration 2, but with a 512KB L2 cache.

Figure 1 shows the normalized performance results when parts of the memory-hierarchy are replaced by their ideal counterparts. This shows that L2 cache misses that take 200+ cycles to get back from memory are a major performance bottleneck (as opposed to L1 cache misses that hit the L2 cache). This makes sense because for our ideal out-of-order processor, the L1 misses that are L2 hits are likely to be tolerated due to the nature of out-of-order execution. Long latency L2 misses, however, can cause the ROB to become full and instruction allocation to stall for a large number of cycles.

In our initial memory profiling study, we made the following prefech-related observations:

1) Using merely the L2 miss addresses observed from the issue stage of an out-of-order processor might not be the best way to train a prefetcher. These addresses may not appear to contain a regular pattern to the prefetcher in certain cases.
2) It is beneficial to use both cache hits and misses of a program to train the predictor's state. By training a prefetcher using only the observed miss addresses, as

[1]We did not skip the first 40 billion instructions for 998.specrand, 999.specrand, and 481.wrf because these programs did not have enough instructions.

[2]Note that bandwidth limitations on the L2 cache and DRAM memory still applied in configurations 2 and 3.

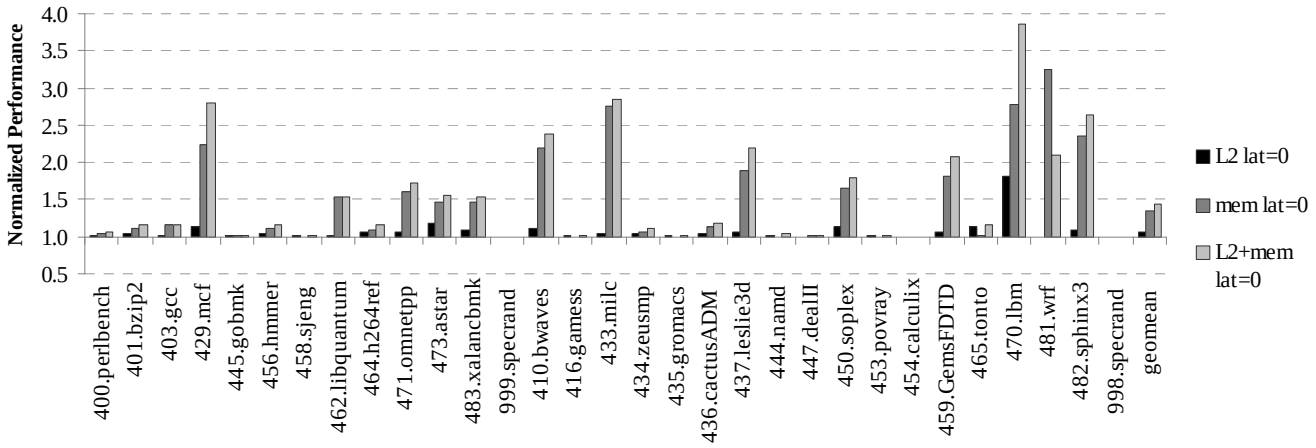

Fig. 1. Simulation results showing normalized performance when different parts of the memory hierarchy are replaced by their ideal counterparts. The performance here is an average over 3 configurations explained in section II. An accurate L2 prefetcher can provide a signifcant portion of the available performance opportunity.

some prior work did, the prefetcher may not be able to detect any useful pattern.

3) It is beneficial to train the prefetcher with both the local (per PC) and global access information. Some access patterns are not visible at the global level, while others may only be visible if the prefetcher organizes them by the program counter (PC).
4) It might be helpful to classify accesses into regions (i.e., spatially by memory addresses) and detect patterns within their respective regions. For example, in 470.lbm and 462.libquantum, the stride can be easily detected if the accesses are binned according to region. This is because a single instruction can access multiple regions within a period of hundreds of cycles. This might be more visible in a CISC architecture because of complex instructions that generate more than a single memory uop (example: the `rep` family of instrutions in x86 decomposes into at least 2 memory uops that move data from one memory location to another).
5) It might be beneficial to trigger a prefetch request upon each cache *access* (regardless of whether it is a hit or miss) rather than only on a cache *miss*.
6) The consecutive memory accesses of some benchmarks follow a multiplicatively increasing stride. For example,the 429.mcf has accesses with these deltas (in terms of cache lines): 6, 13, 26, 52, etc. This pattern may be observed when the program is chasing pointers that are part of a tree data structure (example: going down a heap data structure laid out as an array).
7) It could be useful to generate prefetches based on partial matches, given the logic complexity is not too prohibitive. Sometimes a program may not access memory in a completely regular fashion, however, patterns can still be detected in a fuzzy way.

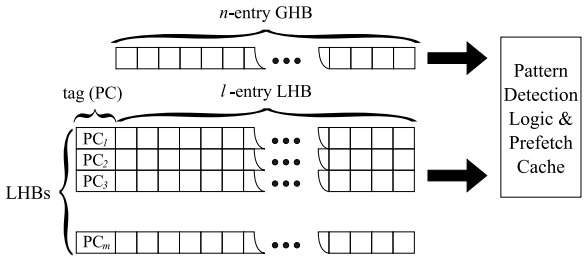

Fig. 2. Block diagram of our proposed prefetcher. Each GHB stores n memory access addresses made by the processor. Each LHB stores, along with the PC value, access addresses made by the instruction at that particular PC. The array of LHBs is a fully associative structure, looked up by PC. For our submission, n=128, l=24 and m=32. The prefetch cache is a 32-entry fully associative structure that stores previously issued prefetches.

III. Prefetcher Design and Implementation

Motivated by the observations from those memory-bound benchmark programs in SPEC, we proposed a data prefetcher design that uses both a Global History Buffer (GHB) and multiple Local History Buffers (LHB). The GHB tracks the most recent n memory accesses (both loads and stores) in a program while each of the LHBs tracks m accesses performed by l a particular PC (each LHB is tagged with a 32-bit PC value). The particular values used for l, n and m in our prefetcher are given in Section IV. The list of LHBs is maintained as a fully associative cache with the LRU replacement policy. The number of valid entries in the GHB or LHB are not stored anywhere – instead, an invalid address `(CacheAddr_t)(NULL)` or `(CacheAddr_t)(-1)` is inserted after the last valid address. A block diagram of our prefetcher is given in Figure 2.

When an address request is generated, the processor looks up the GHB and the LHB. An access address is only added if it is not found in these buffers. If the PC is not found in the list of LHBs, the least-recently-used LHB is selected

Algorithm 1 The repeating pattern detection algorithm. After a repeated pattern is detected, a score can be assigned based on the length of the repetition (the variable labeled `max` in FindRepeatingPattern).

```
1:  procedure FINDREPEATINGPATTERN(deltas, N)
2:      max ← 0
3:      for i ← N − 1, 1 do
4:          currentValue ← CalcRepeatingScore(deltas, i, N)
5:          if currentValue > max then
6:              max ← currentValue
7:              maxI ← i
8:          end if
9:      end for
10:                                  ▷ The length of the pattern repeats is now in max. If this greater than a threshold, prefetch is issued.
11: end procedure

12: procedure CALCREPEATINGSCORE(deltas, start, end)
13:     length ← end − start
14:     totalScore ← 0
15:     currStart ← start
16:     currEnd ← end
17:     while currEnd > 0 do
18:         currentScore ← Compare(deltas, start, end, currStart, currEnd)
19:         totalScore ← totalScore + currentScore
20:         if curentScore < length then
21:             return totalScore
22:         end if
23:         currEnd ← currEnd − length
24:         currStart ← currStart − length
25:     end while
26:     return totalScore
27: end procedure

28: procedure COMPARE(deltas, start, end, currStart, currEnd)
29:     score ← 0
30:     i ← 0
31:     while deltas[currEnd − i] == deltas[end − (i % (end − start))] && currEnd − i >= currStart do
32:         score ← score + 1
33:         i + +
34:     end while
35:     return score
36: end procedure
```

and the access address is stored as the first entry of the LHB (further memory accesses from the same PC will be entered in this particular LHB.). Either buffer, the GHB or a particular LHB, can trigger prefetches when an address it added to it. From these buffers, we have logic that does the following:

1) Inspects the deltas of *accesses* including both hits and misses in the recent past in a 64KB region around the latest access to see if there is a repeatable pattern. If so, prefetches for the next addresses following the pattern might be generated. A brute-force method is employed to find the repeatable pattern in the deltas. This method is shown in Algorithm 1.
   Example: With accesses (in terms of cache line numbers) of: 0x10, 0x12, 0x13, 0x15, 0x16, the deltas are: 2, 1, 2, 1. The prefetcher will detect this repeating pattern of 2, 1 and prefetch 0x18, 0x19, etc.
2) Inspects the deltas in the *L2 misses* in the recent past (if the buffer contains any) in a 64KB region around the latest access. If it can find a repeatable pattern in them, it prefetches the next few addresses.
3) Inspects the deltas of *accesses* in a region close to the latest access and orders them with respect to address space (as opposed to time, which is the default order). It then tries to issue prefetches if there is a repeatable pattern within these deltas and issues.
   Example: With accesses (in terms of cache line numbers) of: 0x11, 0x9, 0x10, 0x12, the sorted deltas are in fact (from 0x9) 1, 1, 1. The prefetcher will detect this unit stride and will prefetch 0x13, 0x14, etc.
4) Inspects the deltas of *accesses* in a region close to the latest access to see if a multiplicative increasing stride is found. If detected, the next few addresses from this pattern are fetched.
   Example: With accesses (in terms of cache line numbers) of: 0x10, 0x12, 0x16, 0x1E, the deltas are: 2, 4, 8. The prefetcher will detect an exponential stride and will prefetch 0x2E, 0x4E, etc.
   Note: Since the prefetcher has access to the *cache line numbers* instead of exact *byte addresses*, slightly inexact matching is used for detecting this pattern. For example,

with deltas (in terms of cache lines) of 6, 13, 26, 53 do not strictly follow an exponential series, they will be detected as such by the prefetcher. If the simulation infrastructure allowed exact byte addresses to be used as input to the prefetcher, this inexact matching would likely not be needed.

In a single cycle, multiple prefetches may be generated (depending on how many memory accesses are made per cycle, among other things). If these prefetches are issued indiscriminately, precious resources such as bandwidth and MSHR entries could be oversubscribed. To prevent multiple prefetches to the same cache line from generating redundant requests, a 32-entry, fully-associative cache is checked before issuing the request.

IV. Parameters and Storage

There are three incarnations of the prefetcher that have been submitted. All three have a GHB size of 128, a LHB size of 24 with 32 entries. The cost of GHB is 128*32 bits = 4096 bits. The difference between these submissions is the aggressiveness of the prefetchers. Specifically, these prefetchers have different limits on the `max` variable given in Algorithm 1 before they issue the prefetches. The cost of LHBs is 32*(24+1)*32=25600 bits.[3] In addition, there is a prefetch cache of 32 entries (32*32bits=1024 bits) to filter out redundant prefetches. Therefore, the total storage cost of the prefetcher is 1024+25600+4096 bits = 30720 bits. This is under the 32KB storage limit not counting the temporary variables. With unlimited hardware complexity, all the temporary variables in the prefetcher can be incorporated as stateless logic.[4]

V. Simulation Methodology

For our simulations, we relied on a standard version of the prefetcher kit provided by the organizing committee of DPC-1 [1]. This kit contained PIN [3] and CMP$im[2] for generating and simulating traces. To generate instructional traces, 40 billion instructions were skipped and a trace 100 million instructions long was obtained.[5] An out-of-order model was used for the processor with a 128-instruction window. Table I details the other parameters of the performance model of the processor.

A prefetcher as explained in Section III was instantiated and was trained every cycle using the callback function available in the simulation infrastructure. This prefetcher was the best performing one amongst the three submitted versions, and was medium in terms of prefetch aggressiveness. Figures 3 and 4 show the simulation results obtained for the SPEC2006 suite for the three different configurations outlined in Section II. As can be seen from the figure, the prefetcher improves SPEC benchmarks on average by about 20% (geometric mean). Some observations seen by us are observed in the following sub-sections.

[3]Our prefetcher just uses the cache line address and not the full address, so we use 32-6=26 bits for addresses. 2 extra bits are used for storing the access type (i.e. L1 vs L2), and storing whether or not it was a hit.

[4]We count temporary variables as variables that do not persist across cycles. These variables are created on-the-fly in the software prefetching code, and are cleared every cycle. The storage cost calculation only includes storage that persists across cycles. For example, our code requires a temporary sorted array of a small region of the GHB. This can be done by using a hardware sorting network and feeding it into the stride detector in a single cycle without storing it anywhere. Therefore, its cost is not included in the cost calculation.

[5]998.specrand, 999.specrand and 481.wrf did not have their first 40 billion instructions skipped because they did not have enough instructions in the program

TABLE I
Table detailing parameters of our simulated CPU.

Parameter	Value	Notes
Front End	Perfect	No fetch hazards
Window Size	128	
Issue Width	4	No restriction other than true dependencies
L1 Cache Latency	0	
L2 Cache Latency	20	
Memory Latency	200	
L1 Cache Size	32 KB	
L1 Cache Assoc.	32	
L2 Cache Size	2MB / 2MB / 512KB	Config 1 / 2 / 3
L2 Cache Assoc.	32	
L2 Cache Bandwidth	1000 / 1 / 1 accesses per cycle	Config 1 / 2 / 3
Memory Bandwidth	1000 / 0.1 / 0.1 accesses per cycle	Config 1 / 2 / 3

A. Benchmarks not limited by the memory sub-system

In the SPEC suite, there are a few benchmarks that are not limited by the memory sub-system according to our initial study. Benchmarks such as 447.dealII and 453.povray are examples that may be limited by the issue width or the number of floating point units on the simulated machine (the front-end of the machine had perfect branch prediction). For these benchmarks, it is desirable for the prefetcher to not interfere with the current cached contents. Our prefetcher does not degrade the performance of these benchmarks.

B. Streaming Benchmarks

Benchmarks like 470.lbm and 462.libquantum have streaming memory behavior. For such applications, since the access patterns are so simple and predictable, our predictor makes accurate predictions and reduces the last-level cache misses by a large percentage for configuration 1. The reason for the high number of LLC misses in configurations 2 & 3 for 470.libquantum is that multiple requests to identical addresses take up multiple slots in the L2 queue, which is a finite resource. This particular benchmark produces a lot of L2 misses to the same address, causing the L2 queue to be full most of the time. Our prefetcher in debug mode reported that most of its prefetch requests were being ignored by the simulator due to the L2 queue being full.

C. Pointer-Chasing Benchmarks

429.mcf is an example of a pointer-chasing benchmark. In at least one of it's phases, it exhibits accesses whose deltas

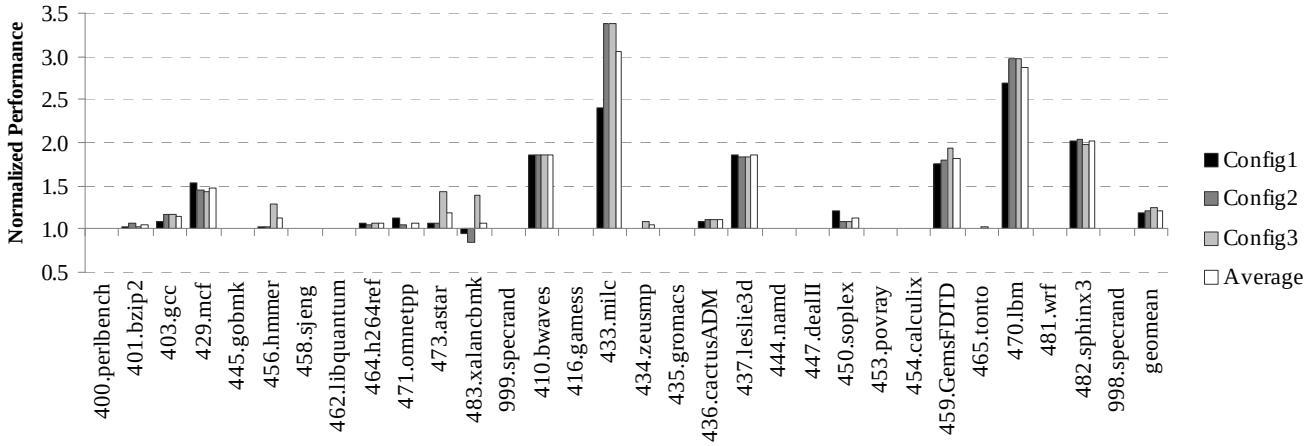

Fig. 3. Performance simulation results of our proposed prefetcher. The average of the three configurations is 20%, as shown by the white bar on the right.

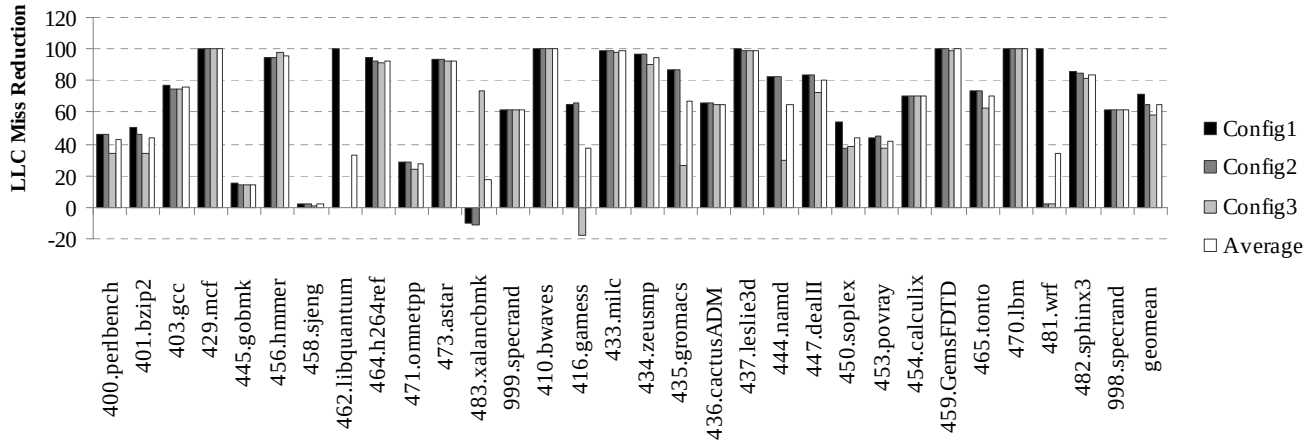

Fig. 4. LLC miss reduction from using our proposed prefetcher. On average, we reduce the LLC misses by over 60%.

follow a geometric pattern. Our prefetcher can reduce a large number of LLC misses by correctly predicting the future access addresses as explained in the previous section.

D. Benchmarks Negatively Affected

One benchmark in the suite, 483.xalancbmk, was negatively affected by our prefetcher in configuration 1 and 2. The performance was reduced by 10% and the LLC misses were actually increased. This is because our prefetcher is too aggressive and often predicts erroneous addresses. These can pollute the cache and eat up into the access bandwidth. Given more storage and access to more microarchitecture modules like MSHRs, etc., we believe that we can improve the performance of benchmarks like 483.xalancbmk. Unfortunately, given the circumstances of the competition and the simulation infrastructure, we did not implement any feedback control to throttle the prefetcher in this case.

VI. Hardware Realization

Although in its present form, the logic and control part of this prefetcher can be costly to implement as-is in hardware, there could be a few changes to the prefetcher that could make it practical. These changes are beyond the scope of this paper, which only seeks to implement the best performing prefetching algorithm with a fixed budget size (thereby ignoring all hardware complexity). During the design of this prefetcher we came up with a few ideas for making the prefetcher more hardware-friendly, but all those schemes took up more storage than what our proposed prefetcher currently uses.

VII. Conclusion

In this paper, we presented a prefetcher that can significantly reduce the number of last-level cache misses (over 60% on average for SPEC) by exploiting and understanding a variety of memory access patterns. It can effectively improve the performance of SPEC suite by about 20% on average.

VIII. Acknowledgments

The authors would like to thank Dong Hyuk Woo for his valuable feedback during the design of the prefetcher. This work was supported in part by an NSF CAREER Award (CNS-0644096).

References

[1] DPC-1 Homepage. 2008. http://www.jilp.org/dpc/.

[2] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proc. Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, Beijing, China, 2008. ACM.

[3] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[4] Kyle J. Nesbit and James E. Smith. Data Cache Prefetching Using a Global History Buffer. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 96, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Cloyce D. Spradling. SPEC CPU2006 Benchmark Tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, 2007.