Pefetching On-time and When It Works

Ibrahim Burak Karsli, Mustafa Cavus and Resit Sendag
Department of Electrical, Computer, and Biomedical Engineering
University of Rhode Island, Kingston, RI, USA
(bkarsli, mcavus, sendag)@ele.uri.edu

Abstract

In this paper, we propose a mechanism that monitors events on the prefetch queues, such as hits, misses and elapsed time between prefetch and demand requests, in order to evaluate timelines of prefetching requests. This information is then fed back to the prefetcher to achieve timely issuing of prefetches. Our prefetcher is further supported by a mechanism to make decisions on completely turning off prefetching when necessary. We show that a simple sequential (next-line) prefetcher supported by our mechanism outperforms the ampm-lite prefetcher and the recently-proposed sandbox prefetching method. Our proposed prefetcher achieves a 4-configuration score of 4.584 on a set of 40 SPEC CPU 2000, SPEC CPU 2006 and Olden benchmarks, while ampm-lite and sandbox prefetchers achieve 4.51 and 4.578, respectively. We further added an ip-stride prefetcher to achieve an overall score of 4.616.

I. INTRODUCTION

Modern processors employ prefetchers to hide long memory latencies for demand cache misses. Prefetchers predict data or instruction addresses those are likely to be used in near future. When successful, they facilitate faster retrieval of data/instruction for demand requests. Next-line or sequential prefetching has been shown to provide significant performance benefits for applications with good spatial locality. However, they prefetch rather blindly because they do not employ confidence mechanisms. This is problematic for two reasons: 1) they use cache and bandwidth resources rather blindly, which can either reduce their benefit, or can even hurt the performance and power; 2) even for the applications with good spatial locality, they may not provide the potential benefits because they are not timely in issuing prefetches. To address the first problem, Pungsley et al. [1] proposed the sandbox prefetching method. In their method, a set of predetermined sequential offset prefetchers are tested by recording their predicted prefetching addresses in the sandbox on each demand access and counting the number of demand access hits on the recorded potential prefetch addresses in the sandbox. After the evaluation interval, only the prefetchers with sandbox scores above a threshold are allowed to perform prefetching in the next interval. The sandbox proves to be a powerful idea eliminating many unnecessary and potentially harmful prefetches – only after a prefetcher has been proven useful, it is activated.

The second problem, although equally important in designing successful prefetchers, is not sufficiently addressed by the sandbox. If prefetch is not timely, there is no benefit. It is possible that a prefetch is issued too late so it does not hide latency sufficiently, or it is issued too early and possibly gets evicted from the cache before it is demand accessed. In the latter case, it could potentially harm the performance by evicting useful cache blocks and using the resources inefficiently. In this paper, we focus on both timeliness and accuracy of a data prefetcher. Because next-line prefetcher is simple, yet proven to be very effective, we propose to use a simple sequential prefetcher with helper mechanisms for guiding it to prefetch on-time. Unlike the sandbox method, however, we use only one sequential prefetcher with one testing buffer instead of many sequential prefetchers with different offsets and their own sandbox buffers to achieve a similar performance.

Our decision engine monitors the testing buffer in the same spirit as the sandbox method but its operation and purpose are quite different (details of which are described in Section 2). After each evaluation period, the decision engine increments or decrements a distance counter to guide the sequential prefetcher in how far ahead a prefetch must be issued in the next interval to be useful. Decision on incrementing or decrementing the distance is based on several factors, such as, the number of demand hits in the TQ, the number of L2 misses and the amount and ratio of demand misses that are hit in the TQ. Within selected intervals, the sequential prefetcher is further evaluated for success by turning off prefetching for a range of test addresses and comparing whether demand accesses within that range have a better cache hit rate than the ones not in that range. If prefetching is proved not successful, the decision engine turns it off by setting the distance to zero. Finally, it is important to note that the prefetcher actively issues prefetches and gets evaluated at the same time using only one single testing buffer.

The remainder of the paper is organized as follows. Section 2 discusses the motivation. Section 3 describes the details of the proposed prefetcher. Section 4 discuses the conditions for distance update decision. In Section 5, we discuss the hardware budget. Section 5 presents the results and Section 6 concludes.

II. MOTIVATION

Our profiling results show that majority of sequential offsets are positive therefore our prefetcher uses only positive distances. We ran a simple sequential prefetcher with constant distances (i.e., offsets) from 1 to 10 to observe the number of benchmarks that performed best for each distance. The results are shown in Table 1. On average, 16 out of 40 benchmarks have their best performance with a distance of 1 (the next-line). When the distance is greater than 6, significantly fewer benchmarks observe their best performances. We also observe that configuration has an effect on the best distance, even for the same benchmark.

Although distance 1 (next-line prefetcher) seems to show the best promise (see Table 1), Figure 1 shows that the best average speedup for 40 benchmarks was obtained with a distance of 3, while distance 4 was a close second. This is because for benchmarks where distance 1 was the best, the performances of distances 3 or 4 were not far off. On the other hand, for a significant number of benchmarks, the benefit of distance 3 or 4 was much larger.

Based on our initial analysis, we conclude that a sequential prefetcher that can adaptively change its distance can capture the behavior in Table 1 and Figure 1. It is important to also mention that adaptive distance is not only important across benchmarks, it is also important within the same benchmark when program behavior changes.

Table 1. The best fixed offsets for 40 benchmarks that we studied. Table shows the number of benchmarks that performed best for each distance and configuration pair. The last row is the average number of benchmarks that performed best using that distance.

	Distances									
Configurations	1	2	3	4	5	6	7	8	9	10
Config 1	14	8	5	5	4	1	1	0	1	1
Config 2	18	6	5	5	0	2	0	1	1	2
Config 3	18	4	4	8	4	1	0	0	0	1
Config 4	14	9	3	4	6	2	1	0	0	1
Average	16	8	6	4	2	2	0	0	1	1

III. SEQUENTIAL PREFETCHER WITH ADAPTIVE DISTANCE (SPAD)

Figure 2 illustrates our proposed sequential prefetcher. We employ a First-In-First-Out (FIFO) Testing Queue (TQ) to record the predicted prefetch addresses and the current cycle. TQ is also used as the

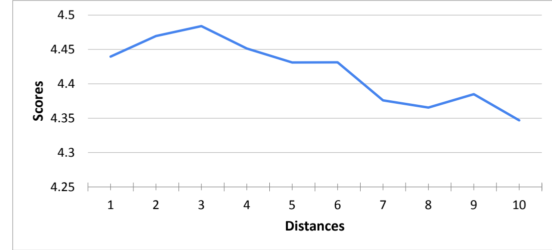

Figure 1. Performance of sequential prefetcher with varying distances. Figure shows 4-configuration scores for fixed distance sequential prefetchers. As can be seen, distance 3 sows the best performance with a 4.48 score.

prefetch filter when prefetcher is active in order to filter prefetch requests going to the memory system. The prefetcher is evaluated every 512 L2 accesses, which is called an *interval*. At the end of each interval, the *decision engine* decrements or increments the distance based on the evaluation at the end of an interval. When the distance is zero, no prefetching would occur in the next interval. However, the TQ continues to record *addr + 1* (next-line) as if a distance 1 prefetcher is active. This is needed to continue evaluating the prefetcher when it is off so that later when the prefetching is useful again, it can get activated. Incrementing or decrementing the distance is based on several factors as given in Section 4.

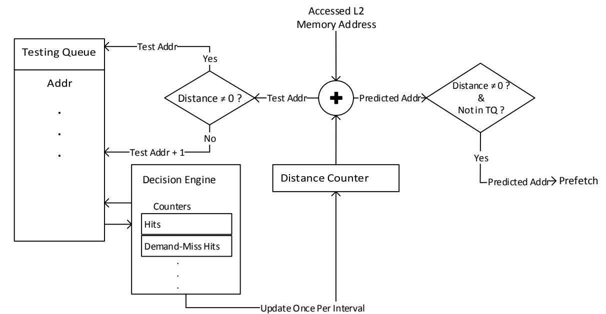

Figure 2. The proposed SPAD prefetcher components

IV. DISTANCE UPDATE DECISION

To make the distance update decision, the decision engine checks several conditions as follows.

1. If the number of demand accesses that are found in TQ, *tqhits*, is less than a threshold (16 in our submission) and the distance is greater than 1 ($tqhits < 16$ && $distance > 1$), distance is decremented.

2. If $tqhits < 16$ for three consecutive intervals, prefetching is considered useless and distance is set to zero disabling the prefetching.

3. If *tqhits >= 16*, the distance update decision is made as follows (in this order):

   a. If the number of L2 misses, *l2miss*, in the interval is less than a threshold (10 in our submission), no update is made assuming current distance value is successful.

   b. If the difference between *l2miss* and the number of L2 misses that were found in TQ, *tqmhits*, is greater than a threshold (0.6 x interval size = 307 in our submission) for more than two consecutive intervals, distance is set to zero disabling prefetching for the next interval.

   c. Finally, *l2miss*/*tqmhits* < 2 for more than two consecutive intervals, distance is incremented.

   If *tqhits* >= 16, and either of the above *a-c* conditions occur more than twice while distance is zero, distance is set to one to turn on the prefetching, which guarantees that prefetching is not off for more than two consecutive intervals.

Furthermore, the decision engine also checks the success of prefetching. This is done by disabling prefetches for test block addresses where *blk_addr* % 4 = 2. Then we check whether or not the cache hit rate of the test addresses is greater than the hit rate of the remaining demand addresses. If it is, the prefetching is turned off by setting the distance to zero. The testing can only occur when prefetching is on. If testing proves prefetcing successful, the current distance between testing intervals is doubled. Since prefetcher continues to record predicted addresses into the TQ when prefetching is off, prefetching can be turned back on if it is proved successful.

V. HARDWARE BUDGET

Table 2 breaks down the hardware cost of the SPAD prefetcher by function and component. The major storage needed for SPAD is for its TQ. In our evaluation, we use a 128-entry TQ. Since SPAD requires very small hardware budget (4327 bits), we did not truncate the addresses and assume each to be 32 bits. SPAD provides good performance with small hardware budget. However, for some benchmarks, an ip-based prefetcher outperforms SPAD. Therefore, we have decided to integrate the ip-stride prefetcher that is provided as an example prefetcher in the competition framework. In this integration, we added a 128-entry global prefetch buffer to filter out the prefetch requests coming from both the SPAD and the ip-stride prefetcher. Overall about 29% of the competition hardware budget was utilized.

Table 2: SPAD hardware cost.

Prefetcher	Components			Budget
Sequential (SPAD)	Prefetch Queue	Address (32 Bit)	128 entries + Tail Pointer (7 Bit)	4103 Bit
	Test Queue	Address (32 Bit)	128 entries + Tail Pointer (7 Bit)	4103 bits
	Registers	L2 Access Counter (32 Bit) L2 Miss Counter (32 Bit) Test Queue Hits Counter (32 Bit) Test Queue Miss Hits Counter (32 Bit) Interval Reg. (32 Bit)		160 bits
Ip Stride	Ip Stride	IP Bit (32 Bit) Last Address Bit (16 Bit) Last Stride Bit (8 Bit) LRU Bit (10 Bit)	1024 Entries	67584 bits
Total (SPAD + ip-stride)				75950 bits
Percentage				29%

VI. RESULTS AND CONCLUSION

In our evaluations, we have used 40 benchmarks from SPEC CPU2000 [3], SPEC CPU2006 [4] and Olden [5] benchmark suites. We use Simpoint 2.0 [6] to generate representative 100M-instruction traces. The results were obtained for each of the four competition configurations: configuration 1 *(no_flags)*, configuration 2 *(small_llc)*, configuration 3 *(low_bandwidth)* and configuration 4 *(scramble_loads)*. For all simulations there were 10 million warmup instructions *(warmup_instructions)* followed by 90 million simulation instructions *(simulation_instructions)*.

Figure 3 shows the speedups compared to no prefetching performed by the SPAD, ip-stride and combined submitted prefetcher, respectively. SPAD improves the performance on four configurations by 16.15%, 16.20%, 10.44% and 15.62%, respectively, to give a total competition score of 4.584. Although this result is significantly better than ip-stride's 4.300 score, ip-stride performed better for a number of benchmarks, most significantly for *bzip2* and *soplex*. SPAD provided no speedup for bzip2, the addition of ip-stride provided 12% speedup for configuration 1. Another very significant speedup change was observed for soplex. While SPAD result in 3.5% speedup, with ip-stride it was 17%. Overall, integrating SPAD with ip-stride improves SPAD performance by 5.5%, on average, giving a score of 4.616. In comparison, AMPM-Lite [7]

had a competition score of 4.511 and our best Sandbox implementation (with 32 offsets (-16 to +16)) earned a score of 4.578.

Figure 4 shows the speedups for each benchmark for the four competition configurations. We can see that, *ammp*, *milc* and *xalanbench* are negatively affected from prefetching. The most significant speedups are obtained for *bwaves*, *zeusmp*, *cactusADM*, *leslie3d*, *hmmer*, *sphinx3*, *health*, *mst*, *perimeter* and *treeadd*.

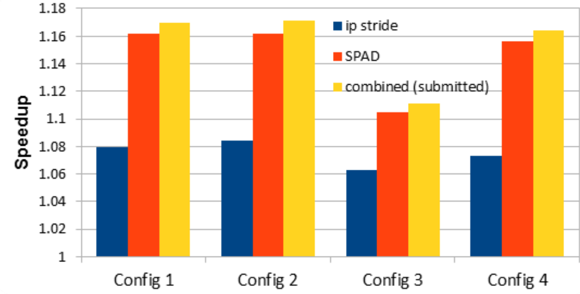

Figure 3: The geometric mean speedup for *ip-stride*, *SPAD* and *combined* prefetchers for configurations 1-4.

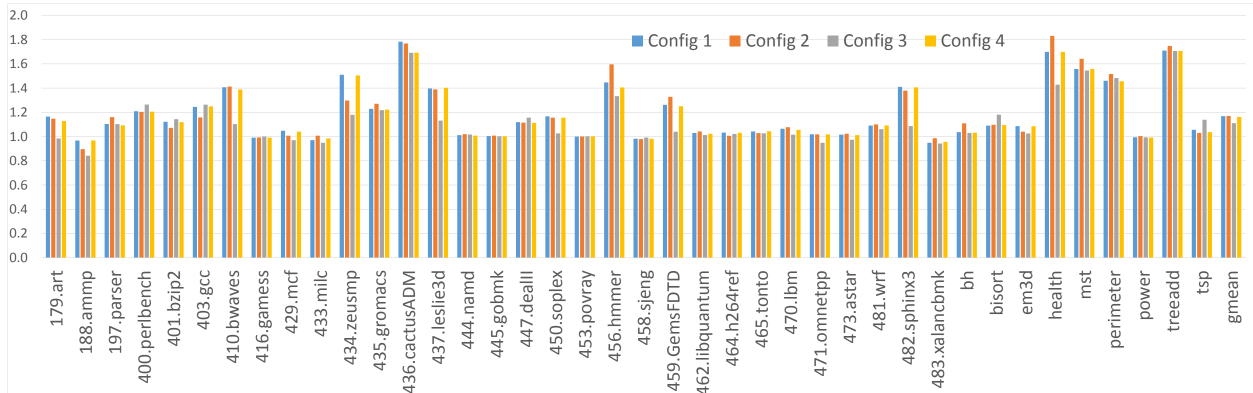

Figure 4. SPAD Performance Results for all benchmarks

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their helpful suggestions. This work is partly supported by the National Science Foundation under grants CCF-1117467, CCF-1422516 and CNS-1405862.

REFERENCES

[1] Steve Vanderwiel and David J. Lilja, "A survey of Data Prefetching Techniques," ACM Surveys, 1996.

[2] Seth Pugsley, Zeshan Chishti, Chris Wilkerson, Troy Chuang, Robert Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, Rajeev Balasubramonian, "Sandbox Prefetching: Safe, Run-Time Evaluation of Aggressive Prefetchers," 20th International Symposium on High-Performance Computer Architecture (HPCA-20) , Orlando, February 2014.

[3] Standard Performance Evaluation Corporation CPU2006 Benchmark Suite. http://www.spec.org/cpu2000/

[4] Standard Performance Evaluation Corporation CPU2006 Benchmark Suite. http://www.spec.org/cpu2006/

[5] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting Dynamic Data Structures on Distributed Memory Machines," ACM Transactions on Programming Languages and Systems, Mar. 1995

[6] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," ASPLOS, 2002.

[7] Yasuo Ishii, Mary Inaba, Kei Hiraki, "Access map pattern matching for data cache prefetch," Proceedings of the 23rd international conference on Supercomputing, 2009, Yorktown Heights, NY, USA, June 8-12, 2009. Also in 1st Championship Data Prefetching Competition. http://www.jilp.org/dpc/online/papers/03ishii.pdf