Sandbox Based Optimal Offset Estimation

Nathan T. Brown and Resit Sendag
Department of Electrical, Computer, and Biomedical Engineering
University of Rhode Island, Kingston, RI, USA
(nattayb, sendag)@ele.uri.edu

Abstract

In this paper, we introduce a prefetching mechanism that dynamically estimates optimal sequential offset based on accuracy and timeliness. Our prefetcher is based on the recently-proposed sandbox prefetching method. The original sandbox prefetching utilizes multiple sequential prefetchers, which are evaluated based on theoretical prefetches to identify the highest performing offsets. However, problems arise when a benchmark requires a large prefetch distance for prefetching to be useful. We modified the offset scoring of the sandbox to take into account the estimated fill-time, which adds another evaluation dimension to the sandbox scores by adjusting the prefetchers with inadequate distance. The coupling of both accuracy and timeliness estimation reveals the most efficient sequential prefetcher. The highest corrected scoring prefetcher is then gated by a threshold. If the corrected score exceeds the requirement, it is then promoted to live status for the next evaluation period.

I. Introduction

The purpose of a data prefetcher is to predict the blocks of memory which will be demanded by the given application in the future. In this way the number of cache misses is minimized throughout the memory hierarchy improving the performance. While there are a variety of unique algorithms and techniques for accomplishing this, no single approach will succeed on all processor architectures and/or for all applications. An assortment of hazards plague prefetchers based on the size, bandwidth and cache inclusion policies for multi-level cache architectures. This proposal aims to expand upon the benefits of *Sandbox Prefetching* [1] by dynamically selecting the live prefetcher based not only on accuracy but also timeliness. A mechanism is employed to track the real-time latency of requests to the lower level memories (*L3 and Main Memory*) in an effort to track when lines are filled.

II. Related Work

A. Sequential Prefetching

Arguably one of the simplest and yet very effective prefetchers is the *Sequential Prefetcher*. This type of prefetcher takes advantage of the spatial locality many applications exhibit when accessing memory. Prefetch predictions are determined by adding a fixed negative or positive *Offset* to the accessed address. In the case of a next line prefetcher a stream of accesses such as $A_1, A_2 \ldots A_n$ will yield a minimum of one prefetch to the addresses $A_2, A_3 \ldots A_{N+1}$. A prefetching *Distance* or *Degree* can be added to increase the timeliness and/or the aggressiveness. While Sequential prefetchers can produce significant speedups, one of the inherent problems is that each benchmark will have different access patterns which cater to a given offset, distance, and degree combination. In a hostile benchmark the lack of confirmation will significantly deteriorate the performance. The result is that a single prefetcher will rarely perform well over a wide array of applications.

B. Sandbox Prefetching

This type of prefetching system utilizes a real-time confirmation based system to evaluate a series of unique-offset sequential prefetchers. Proposed by Pugsley et al. [1], the *Sandbox* is a trial area for which candidate prefetchers make predictions on each cache access. On each cache access the sandbox is tested to see if it contains the address. If there is a hit in the sandbox, the score for the prefetcher is increased. At the end of the evaluation period the sandbox is reset and a new prefetcher is evaluated. Following the round robin evaluation, prefetchers are allowed to make prefetches based on graded thresholds with priority given to the more immediate offsets.

This approach identifies the most accurate prefetchers, meaning those which are correct the most over a period of time. However, while a given prefetcher may be the most accurate, it might not be far enough ahead of the impending demand request to benefit fully.

III. Sandbox Based Optimal Offset Estimation

The principle of *Sandbox Based Optimal Offset Estimation* is to evaluate candidate prefetchers to determine which produces the most accurate prefetches that are filled prior to their demand access. Identification of the prefetcher with the optimum offset enables less overall prefetches to be issued while at the same time reducing the risk of cache pollution. SBOOE expands upon the scoring system of *Sandbox Prefetching* to evaluate candidate prefetchers based on their accuracy,

as well as their timeliness. Collectively, the two parameters can be utilized to create a measure of a prefetchers usefulness to the system.

A. Memory Access Latency Estimation

In order to track when simulated prefetches would arrive in the cache it is necessary to determine the latency for an access to the L3 cache or main memory. Figure 1 illustrates how this is accomplished. 1) Demand misses are tracked and tagged with the cycle at which the miss occurred. 2) When the address is filled in the L2 cache, the elapsed cycles are calculated using the difference between the current cycle and the tag. 3) This instantaneous latency is then shifted into a *First-In First-Out (FIFO)* buffer. 4) When the FIFO reaches maximum occupancy an average is calculated each time a new latency entry is added. This is done by summing the latencies (which can be done using simple two operations: subtract the oldest from the current sum and add the newest), and 5) then shifting them by log_2*(Latency Buffer Size)*. This real-time measurement is used as an approximation of the turnaround time for a given prefetch.

B. Sandbox Implementation and Scoring

Each sandbox entry is composed of two fields: The first is the cache line address predicted by the candidate prefetcher. The second is the *Cycles to Arrival* field which is used to determine the remaining cycles until the simulated prefetch would be filled in the L2 cache. The sandbox itself is composed of *n* total entries where *n* is the number of accesses in the evaluation period. The sandbox structure is organized as a FIFO queue.

The implementation also contains three registers for maintaining scores which are later used for evaluation. The following define each of the three:

The *Sandbox Score* details the number of L2 accesses during the evaluation period which were present in the sandbox. This score essentially represents the theoretical accuracy of the prefetcher.

The *Late Score* contains the number of hits in the sandbox which wouldn't have been filled yet in the cache. This is determined by evaluating whether the *Cycles to Arrival* field was non-zero at the time of the access. This represents the timeliness or lack thereof of the prefetcher.

Finally, the *Useful Score* is not calculated each access but rather upon the completion of the sandbox evaluation period. The value is the *Sandbox Score* adjusted by the *Late Score* ($S_S - S_L = S_U$) and represents a balance between accuracy and timeliness.

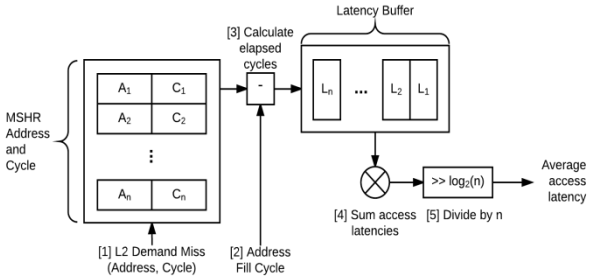

Figure 1: Lower level memory access latency estimation.

Each L2 access, the sandbox is tested to determine if the access was present. If a sandbox hit occurs the *Sandbox Score* is incremented by one and the *Cycles to Arrival* field of the identified address is evaluated. If the remaining number of cycles is non-zero indicating it theoretically would not yet have arrived in the cache then the *Late Score* is incremented by one, otherwise nothing happens. This process is repeated each access while the sandbox occupancy is between 1 and *n*, until *n* accesses have each been verified. Upon the completion of the evaluation period the *Useful Score* is calculated.

C. Architecture and Evaluation

Unlike *Sandbox Prefetching*, SBOOE utilizes a separate sandbox for each sequential prefetcher. While this significantly increases the storage requirement it allows each prefetcher to be evaluated during the same period. Since all the sandboxes are synchronized in respect to accesses and predictions, the completion of the sandbox evaluation period signals the prefetcher score evaluation. During the process, the prefetcher with the highest *Useful Score* is identified. The determined prefetcher's *Sandbox Score* is then compared to a threshold which is established as a quarter of the maximum possible score *n*. This value was identified as optimal to stop prefetching during periods of uncertainty. If the threshold is exceeded the prefetcher makes a prediction, otherwise no prefetch is made. A single prefetch is the maximum allowed for any prefetcher stimulus (L2 cache access).

Figure 2 provides an example demonstrating the utility of estimating the arrival time of issued prefetches. The results are from a single evaluation period (1024 L2 accesses) during a *gcc* simulation. The two highest scores (*UsefulScore + LateScore = SandboxScore)* are the +4 and +8 offset prefetchers. This strongly indicates the stride is 4. However, while the sandbox score is significantly greater than the +4 prefetcher, once it is adjusted for timeliness it is evident that the +8 prefetcher is a more useful choice overall. If a single prefetch is to be made, the +8 prefetcher will ideally yield a greater amount of cache hits.

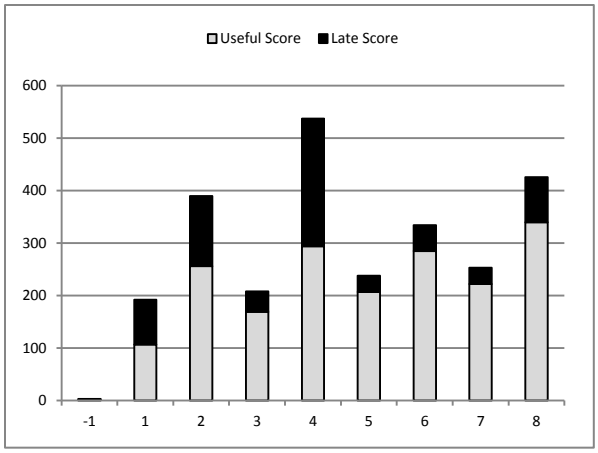

Figure 2: Single evaluation period: *useful, late* and *sandbox* score (*useful* + *late*).

IV. PREFETCHING FRAMEWORK

Collectively, this proposal implements a series of sequential prefetchers with varying offsets and their corresponding sandboxes. Through experimentation a sandbox size of 1024 entries was found to be optimal. The number of candidate prefetchers was driven by hardware storage limitations. Since the rate at which L2 misses occur varies wildly by benchmark, the number of latency samples utilized for each average had an important effect on performance. A slowly updating average latency yields inaccuracies which directly affect the candidate prefetcher scoring. Through testing a size of 32 averages was identified as the best period.

A. Sequential Prefetchers

Nine sequential prefetchers with offsets between -1 and +8 are implemented along with their corresponding sandboxes and supporting resources.

B. Miss Status Holding Registers

While the competition simulator includes a MSHR with 16 registers it only allows the prefetcher access to only the occupancy but not the contents. To overcome this challenge, and to aid with the *Lower Level Memory Latency* estimation a user MSHR of the same size as the system MSHR was implemented. This was done to reduce the chance of secondary misses.

C. Prefetch Buffer

A 64 entry prefetch buffer was implemented to filter potential prefetches. The buffer covered prefetches issued to both the L2 and L3 (LLC) cache. They are removed when the FIFO is full and a new prefetch is issued.

V. HARDWARE AND COMPETITION REQUIREMENTS

The following breaks down the hardware cost of the SBOOE prefetching framework by function and component. To conserve storage space for the buffers the 64 bit byte addresses are first converted to cache line addresses and then truncated to 16 bits for storage. The MSHR cycle buffer stores the lower 16 bits of the processor cycle. The *Cycles to Arrival* tag in each sandbox entry is 10 bits. The extra field in Table1 refers to head and tail indices. The MSHR address and cycle buffer can share them since they're synchronized as can the sandbox address and cycles to arrive buffer. Overall 92.56% of the hardware budget was utilized.

Table 1: SBOOE hardware cost.

Component	*Hardware*				
	Part	*Size*	*Width*	*Extra*	*Cost*
Access Latency Estimation	MSHR Address Buffer	16	16 + 1 bits	8 bits	280 bits
	MSHR Cycle Buffer	16	16 bits	0 bits	256 bits
	Latency Buffer	32	10 bits	10 bits	330 bits
	Last L2 Access Cycle	1	16 bits	0 bits	16 bits
	Average Latency Cycles	1	16 bits	0 bits	16 bits
Prefetch Buffer	Prefetch Buffer	64	16 bits	12 bits	1036 bits
Sequential Prefetchers	Prediction Registers (9)	1	64 bits	0 bits	576 bits
	Primary Index	1	4 bits	0 bits	4 bits
Sandboxes	Address Buffer (9)	1024	16 bits	10 bits	147,546 bits
	Cycles to Arrival Buffer (9)	1024	10 bits	0 bits	92,160 bits
	Sandbox Score (9)	1	16 bits	0 bits	144 bits
	Late Score (9)	1	16 bits	0 bits	144 bits
	Useful Score (9)	1	16 bits	0 bits	144 bits
Total					**242,652 bits**
Percentage					**92.56%**

VI. RESULTS AND CONCLUSION

Figure 3 displays the speedups compared to no prefetching performed by the SBOOE prefetcher for 40 benchmarks from SPEC CPU2000 [2], SPEC CPU2006 [3] and Olden [4] benchmark suites. Representative 100M-instruction traces were generated using Simpoint 2.0 [5]. The results were obtained for each of the four competition configurations: configuration 1 *(no_flags)*, configuration 2 *(small_llc)*, configuration 3

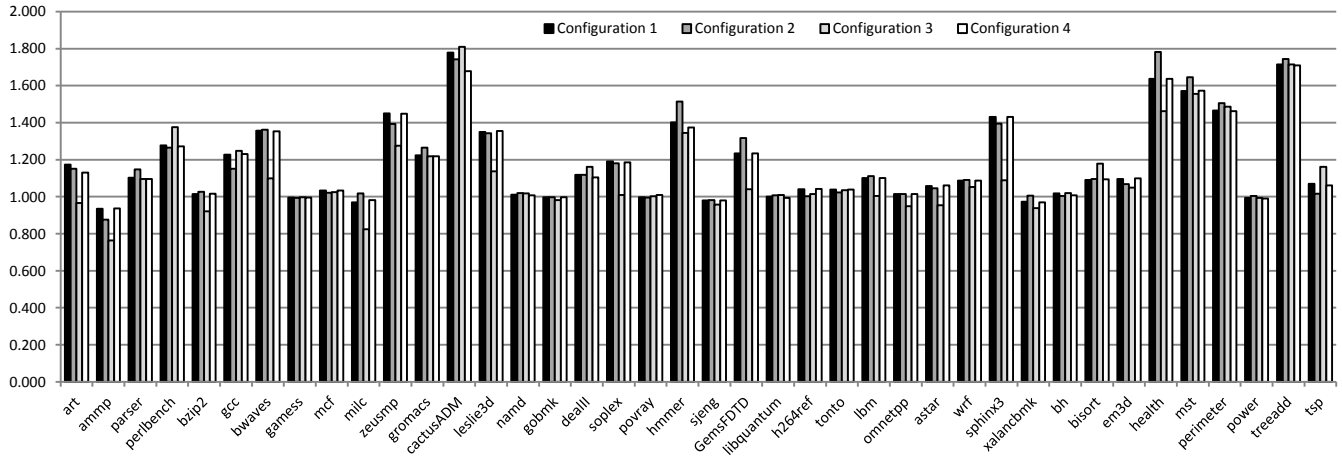

Figure 3: Benchmark speedups by configuration.

(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)*. Performance for the benchmarks was improved by a peak of 16.5% on configuration 2, a minimum of 10.4% on configuration 3 for a total competition score 4.589. In comparison, AMPM-Lite had a competition score of 4.511 and a pure Sandbox implementation earned a score of 4.578. Several benchmarks consistently performed poorly across configurations most notably: *ammp*, *milc* and *xalancbmk*. However, when the individual sequential prefetchers (-16 to +16) were run on the unfavorable benchmarks only +1 was ever able to have any positive effect on performance. This suggests that most likely the applications do not follow a stride very often and would potentially benefit from a PC based prefetcher with confirmation or other context-based prefetchers.

In conclusion, measuring the fill latency for demand misses has proved to be an effective measurement of lower level memory access latency. When coupled with a real-time evaluation system like *Sandbox Prefetching* the evaluation of both accuracy as well as timeliness proves to be effective.

VII. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful suggestions. This work is partly supported by the National Science Foundation under grant CNS-1405862.

REFERENCES

[1] 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.

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

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

[4] 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

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