Appears in the Third Data Prefetching Championship (DPC3), in conjunction with International Symposium on Computer Architecture (ISCA), 2019

Multi-Lookahead Offset Prefetching

Mehran Shakerinava[‡] Mohammad Bakhshalipour[‡] Pejman Lotfi-Kamran[§] Hamid Sarbazi-Azad[‡§]

[‡] Department of Computer Engineering, Sharif University of Technology

[§] School of Computer Science, Institute for Research in Fundamental Sciences (IPM)

Abstract

Offset prefetching has been recently proposed as a low-overhead yet high-performance approach to eliminate data cache misses or reduce their negative effect. In offset prefetching, whenever a cache block (e.g., A) is requested, the cache block that is distanced by k cache blocks (e.g., $A + k$) is prefetched, where k is the *prefetch offset*. This type of data prefetching imposes minimal storage overhead and has been shown quite effective for many important classes of applications.

In this work, we find that prior proposals for offset prefetching either neglect timeliness or sacrifice miss coverage for timeliness when choosing the prefetch offset. To overcome the deficiencies of prior offset prefetchers, we propose Multi-Lookahead Offset Prefetcher (MLOP), a new mechanism for offset prefetching that considers both miss coverage and timeliness when issuing prefetch requests. MLOP, like prior proposals, evaluates several offsets and allows the qualified offsets to issue prefetch requests; however, unlike them, MLOP *considers multiple prefetching lookaheads during the evaluation of prefetch offsets*. MLOP uses a lightweight hardware structure, composed of a small storage and a simple logic, to *identify the prefetching offsets that could have covered a specific cache miss with various prediction lookaheads*. Based on this, MLOP assigns scores to the prefetch offsets and for each lookahead, selects the highest scoring offset for issuing prefetch requests. We evaluate and compare MLOP with various recent state-of-the-art data prefetchers and show that our proposal improves system performance by 30% over a system with no data prefetcher and by 4% over the previous best-performing data prefetcher.

1 Introduction

Data prefetching has been long proposed and adopted to overcome the performance penalty of long latency cache misses. By predicting the application's future memory accesses and fetching those that are not in the on-chip caches, data prefetchers significantly hide the latency of memory accesses, thereby increasing system performance.

Traditionally, the ability of data prefetchers at enhancing the performance was the single metric at evaluating data prefetchers. As such, prefetchers have grown in their performance benefits, while other factors such as imposed overheads have been marginalized. However, with the widespread use of multi- and many-core processors, and accordingly, the movement towards *lean cores* [3], computer architects re-design nearly all components, considering low overhead as a major design constraint. One of the components that has recently been targeted for simplification is the data prefetcher. Recent research [6, 12, 13, 15, 16] advocates the use of simple and low-overhead data prefetchers, even if they offer slightly lower performance compared to high-performance but extremely-high–overhead prefetcher designs.

The research towards *lean data prefetchers* has culminated in *offset prefetching* [15, 16]. Offset prefetching, in fact, is an evolution of stride prefetching, in which, the prefetcher does *not* try to detect strided streams. Instead, whenever a core requests for a cache block (e.g., A), the offset prefetcher prefetches the cache block that is distanced by k cache lines (e.g., $A + k$), where k is the *prefetch offset*. In other words, offset prefetchers do not correlate the accessed address to any specific stream; rather, they treat the addresses individually, and based on the prefetch offset, they issue a prefetch request for every accessed address. Offset prefetchers have been shown to offer significant performance benefits while imposing small storage and logic overheads [15, 16].

The initial proposal for offset prefetching, named Sandbox Prefetcher (SP) [16], attempts to find offsets that yield *accurate* prefetch requests. To find such offsets, SP evaluates the prefetching accuracy of several predefined offsets (e.g., $-8, -7, \ldots, +8$) and finally allows offsets whose prefetching accuracy are beyond a certain threshold to issue actual prefetch requests. The later work, named Best-Offset Prefetcher (BOP) [15] tweaks SP and sets the *timeliness* as the evaluation metric. BOP is based on the insight that accurate but *late* prefetch requests do not accelerate the execution of applications as much as timely requests do. Therefore, BOP finds offsets that yield timely prefetch requests in an attempt to have the prefetched blocks ready before the processor actually asks for them.

In this work, we take another step and propose a novel offset prefetcher. We observe that while the state-of-the-art offset prefetcher is able to generate timely prefetch requests, it loses much opportunity at covering cache misses because of *relying on a single best offset and discarding many other appropriate offsets*. The state-of-the-art offset prefetcher (BOP)

evaluates several offsets and considers the offset that can generate the most timely prefetch requests as the best offset; then, it relies *only* on this best offset to issue prefetch requests until another offset becomes better, and hence, the new best. In fact, this is a binary classification: the prefetch offsets are considered either as *timely* offsets or *late* offsets. After classification, the prefetcher does *not* allow the so-called late offsets to issue any prefetch requests. However, as we discuss in this paper, there might be many other appropriate offsets that are less timely but are of value in that they can hide a significant fraction of cache miss delays.

To overcome the deficiencies of prior work, we propose to have a *spectrum of timelinesses* for various prefetch offsets during their evaluations, rather than binarily classifying them. During the evaluation of various prefetch offsets, we consider *multiple lookaheads* for every prefetch offset: *with which lookahead can an offset cover a specific cache miss?* To implement this, we consider several lookaheads for each offset, and assign score values to every offset with every lookahead, *individually*. Finally, when the time for prefetching comes, we find the *best offset for each lookahead* and allow it to issue prefetch requests; however, the prefetch requests for smaller lookaheads are *prioritized* and issued first. By doing so, we ensure that we allow the prefetcher to issue enough prefetch requests (i.e., various prefetch offsets are utilized; high miss coverage) while the timeliness is well considered (i.e., the prefetch requests are ordered). Putting all together, we propose the Multi-Lookahead Offset Prefetcher (MLOP), a novel offset prefetcher, and show that it significantly improves system performance over prior state-of-the-art data prefetchers. Through a detailed evaluation of a set of 57 single- and multi-core workloads, we show that MLOP improves system performance by 30% over a baseline with no data prefetcher and by 4% over prior state-of-the-art data prefetcher.

2 The Proposal

Figure 1 shows an overview of our proposal. To extract offsets from access patterns, we use an *Access Map Table (AMT)*. The *AMT* keeps track of several recently-accessed addresses, along with a bit-vector for each base address. Each bit in the bit-vector corresponds to a cache block in the neighborhood of the address, indicating whether or not the block has been accessed. For keeping track of recent accesses, this mechanism (i.e., base address plus bit-vector) works better than storing full addresses in terms of storage efficiency, since accesses exhibit significant spatial localities[1]. We size the bit-vectors to embrace 64 bits.

Like prior proposals [15, 16], we consider an evaluation period in which we evaluate several prefetch offsets and choose the qualified ones for issuing prefetch requests later

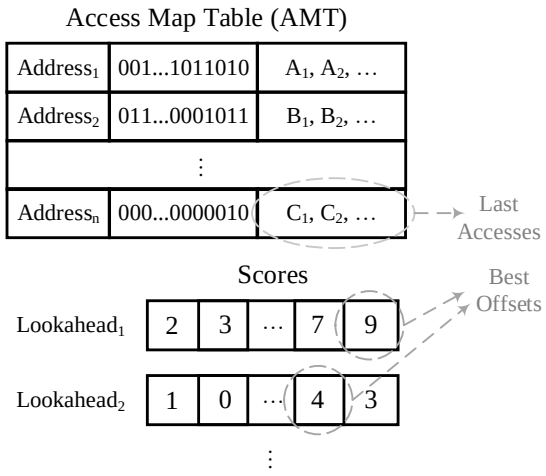

Figure 1. The hardware realization of our proposal.

on[2]. For every offset, we consider multiple levels of score where each level corresponds to a specific lookahead. That is, at the end of the evaluation period, we would have a vector of scores per lookahead level: prefetch offsets at lookahead level one would have their own scores, independent of offset scores at lookahead level two.

The score of an offset at lookahead level X indicates the number of cases where the offset prefetcher could have prefetched an access, at least X accesses prior to occurrence. For example, the score of offsets at the lookahead level 1 indicates the number of cases where the offset prefetcher could have prefetched any of the futures accesses. As the lookahead level increases (say, at lookahead level 10), we approach the case where the prefetcher has *enough* time to issue the prefetch request (i.e., the access that we are attempting to prefetch would happen at least 10 accesses in the future); conversely, at lookahead levels close to 1, the prefetcher does *not* have much time, because the corresponding access would often happen shortly, within a few accesses. We set the number of lookahead levels to 16, efficiently trading off between the imposed overheads (e.g., metadata storage) and performance values.

To efficiently mitigate the negative effect of all predictable cache misses, we select *one best offset from each lookahead level*. Then, during the actual prefetching, we allow all selected best offsets to issue prefetch requests. Doing so, we ensure that we choose enough prefetch offsets (i.e., do not suppress many qualified offsets like prior work [15]), and will cover a significant fraction of cache misses, that are predictable by offset prefetching. To handle the timeliness issue, we try to send the prefetch requests in a way that the application would have sent if there had not been any prefetcher:

[1]This mechanism is also employed by pieces of prior work in data prefetching [10] and even instruction prefetching [9, 14] literature.

[2]The length of the evaluation period is directly tied to the mechanism and components of methods and varies from one method to another. We empirically found that 500 accesses is a suitable evaluation period length for our method.

we start from lookahead level 1 (i.e., the accesses that are expected to happen the soonest) and issue the corresponding prefetch requests (using its best offset), then go to the upper level; this process repeats. With this prioritization, we try to hide the latency of all predictable cache misses, as much as possible.

To update offset scores at lookahead level 1, whenever an access occurs, we find its corresponding bit-vector (using its high-order bits to search the AMT), and then based upon the bit-vector information, we identify the offsets that could have prefetched this access[3], and accordingly, increase the score of those offsets for the first lookahead level[4]. Finally, we set the bit that corresponds to the currently-accessed block in the AMT.

To update the score values in lookahead level 2 and above, however, we cannot merely rely on the bit-vector, and we need information about the *order* of accesses. That is, if we want to evaluate whether an offset could have prefetched an access with a lookahead of 2, we need to know the *previous access* and exclude it from the evaluation. To enable the evaluation of offsets at lookahead levels higher than 1, up until N, we need to *keep the order of the last* $N-1$ *accesses within each bit-vector*. Therefore, since we consider 16 lookahead levels in our configuration, we hold the last 15 accesses, with the order, within each bit-vector. By keeping track of the last accesses, we can easily exclude some of them from the bit-vector when evaluating offsets in various lookahead levels. For example, when we need to update the offset scores at lookahead level 4, we exclude the bits that correspond to the last three accesses from the bit-vector and then update the score values in the same manner as updating the offset scores at the first lookahead level.

Through a storage sensitivity analysis, we find that a 256-entry (per-core) *AMT* (~8.47 KB) offers a near-optimal performance improvement. The total storage requirement of the prefetcher, including all metadata structures, is less than 12 KB, well fitting in DPC3's rules[5].

3 Evaluation

3.1 Methodology

We evaluate our proposal in the context of the simulation framework provided with DPC3. We follow the evaluation methodology of the championship and run simulations for all 46 provided single-core traces. Out of 46 provided traces, we exclude two memory-insensitive ones, then create 11 random MIX traces from the other 44 (the MIXes are completely different from each other; no single-core trace repeats in two of the MIXes). For better readability, we only report the simulation results for workloads whose performance is highly affected by the evaluated prefetchers, as well as the average of all simulated workloads. We compare our proposal against prior state-of-the-art data prefetchers: BOP [15], ASP [11], and SPP [13].

Best-Offset Prefetcher (BOP) [15] is the state-of-the-art offset prefetcher, as well as the winner of DPC2. On each access, BOP tests a single offset to determine whether it would have been able to predict the current access. BOP uses a direct-mapped structure, named *Recent Requests (RR) table*, to keep track of recently-accessed cache blocks. The size of the *RR* table is purposely chosen to be small so that old cache blocks are automatically replaced by the information of recently-accessed ones.

Aggregate Stride Prefetcher (ASP) is a prefetching mechanism proposed in Jain's Ph.D. thesis on "Exploiting Long-Term Behavior for Improved Memory System Performance" [11]. We include ASP as there are similarities between our mechanism and that of ASP (cf. Section 2). ASP employs a *History Buffer* to extract the qualified offsets, i.e., the offset that can correctly predict a cache miss, thereby issuing prefetch requests based on that information. Moreover, ASP ignores several recent accesses (e.g., eight), which causes the prefetcher to train on accesses that are (temporally [2]) further away, and thus, issue timely prefetch requests.

Signature Path Prefetcher (SPP) [13] is a recent state-of-the-art prefetcher. SPP works based on the signatures that it creates and associates to various access patterns. The main contribution of SPP is that it adaptively adjusts (increases or decreases) its prefetching degree, trying to issue timely prefetch requests while preventing memory bandwidth pollution. To keep the track of metadata information (e.g., signatures), SPP uses a *Signature Table*, which directly influences the ability of the prefetcher at keeping the history of accesses and hence predicting future patterns.

For all prefetching methods, we enlarge metadata tables to the extent that either the performance improvement of the prefetcher plateaus or DPC3's rules are violated. All prefetchers sit in the L1 data cache and are trained by L1-D miss streams.

3.2 Results

Figure 2 and 3 show the miss coverage and performance results, respectively. We report miss coverage results only for single-core programs and performance results for all simulated workloads. Miss coverage is the percentage of cache misses that is covered by the prefetcher. We use Instruction-Per-Clock (IPC) as the performance metric and report the

[3]Every offset, say, k, could have prefetched the currently accessed block, say, A, if the bit corresponding to $A-k$ is set in the bit-vector.

[4]The operation of increasing the score of appropriate offsets can be done in a single cycle by shifting the bit-vector. Due to space limitation, we do not discuss the implementation of this component, further, and refer the reader to prior work [11], where the implementation details of *Aggregate Stride Prefetcher (ASP)* has been discussed.

[5]Note that this storage is chosen to get a near-optimal performance improvement from the prefetcher in the context of DPC3. Our evaluations show that, with storage far below than this level, e.g., 4 KB, MLOP is still able to offer a significant fraction of its optimal performance improvement.

performance improvement of all prefetchers over a system without data prefetcher.

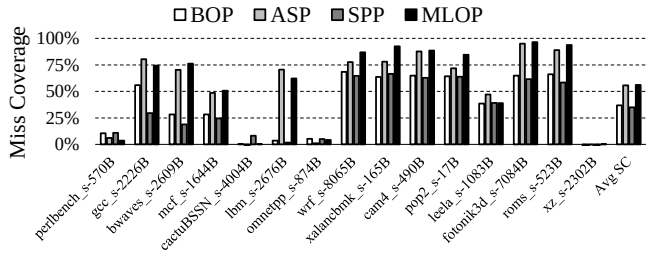

Figure 2. Miss coverage of prefetching techniques. 'Avg SC' stands for the average of all single-core workloads.

MLOP offers the highest miss coverage and performance improvement among the evaluated data prefetchers. On average, MLOP covers 56% of cache misses, which is slightly better than the miss coverage of ASP, the second best-performing method in this regard[6].

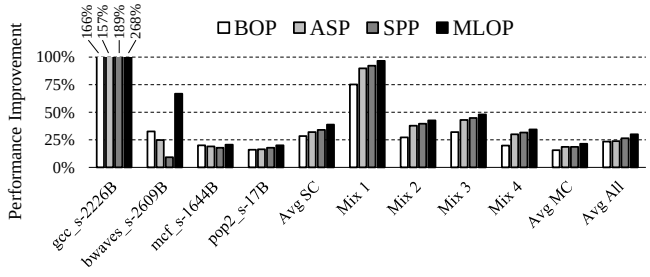

Figure 3. Performance comparison of prefetching techniques, normalized to a baseline system with no prefetcher. Mix1={mcf_s-472B, gcc_s-1850B, roms_s-1070B, cam4_s-490B}, Mix2={xz_s-2302B, mcf_s-484B, fotonik3d_s-8225B, bwaves_s-1740B}, Mix3={pop2_s-17B, roms_s-1613B, bwaves_s-2609B lbm_s-4268B}, and Mix4={roms_s-1007B, fotonik3d_s-7084B, mcf_s-1554B, xalancbmk_s-165B}. 'Avg SC/MC/All' stands for the average of single-core/multi-core/all workloads.

The performance analysis shows that MLOP outperforms the competing prefetching techniques on both single-core and multi-core platforms. On average, MLOP improves performance by 30%, outperforming the second best-performing method (SPP) by 4%.

Miss coverage and timeliness are the main contributors to MLOP's superior performance improvement. BOP, as discussed in this paper, due to its binary classification, neglects many appropriate prefetch offsets and hence, falls short of covering a significant fraction of cache misses. Moreover, we find that another deficiency of BOP arises from the fact that it updates merely a single offset in each update; whereas, our approach, as well as prior proposals like ASP, use vector operations to efficiently update all offset scores at once. ASP, on the other hand, uses a sequential structure to look up the access maps and shifts the entries to create new access maps. We find that this approach suffers from inaccuracy in that frequent shiftings cause the loss of a lot of useful data. Furthermore, ASP adopts a *single global* lookahead (eight) for its distance selections and simply multiplies the prefetching offset to increase the prefetching degree, which usually, based upon our observations, results in a high overprediction rate. Corroborating recent work [4, 5], we find that SPP suffers from the fact that its miss coverage and timeliness is further dependent on the accuracy of its throttling decisions: whenever the throttler makes a wrong prediction, both miss coverage and timeliness of the prefetcher are impaired. MLOP, by considering both miss coverage and timeliness at evaluation and selection of its prefetch offsets, provides the best of both worlds, significantly improving miss coverage and timeliness of prefetching, thereby providing significant performance benefits.

[6]Note that the high miss coverage of ASP comes at the cost of huge overpredictions that it produces (~31% overprediction rate more than MLOP; not shown in the results due to space limitations), which causes memory bandwidth pollution, impairing its performance improvement, especially in multi-core substrates where bandwidth is a scarce resource [1, 7, 8].

References

[1] M. Bakhshalipour *et al.*, "Die-Stacked DRAM: Memory, Cache, or MemCache?" *arXiv preprint arXiv:1809.08828*, 2018.

[2] M. Bakhshalipour *et al.*, "Domino Temporal Data Prefetcher," in *HPCA*, 2018.

[3] M. Bakhshalipour *et al.*, "Fast Data Delivery for Many-Core Processors," *IEEE TC*, 2018.

[4] M. Bakhshalipour *et al.*, "Accurately and Maximally Prefetching Spatial Data Access Patterns with Bingo," *The Third Data Prefetching Championship*, 2019.

[5] M. Bakhshalipour *et al.*, "Bingo Spatial Data Prefetcher," in *HPCA*, 2019.

[6] M. Bakhshalipour *et al.*, "Evaluation of Hardware Data Prefetchers on Server Processors," *ACM CSUR*, 2019.

[7] M. Bakhshalipour *et al.*, "Reducing Writebacks Through In-Cache Displacement," *ACM TODAES*, 2019.

[8] P. Esmaili-Dokht *et al.*, "Scale-Out Processors & Energy Efficiency," *arXiv preprint arXiv:1808.04864*, 2018.

[9] M. Ferdman *et al.*, "Proactive Instruction Fetch," in *MICRO*, 2011.

[10] Y. Ishii *et al.*, "Access Map Pattern Matching for Data Cache Prefetch," in *ICS*, 2009.

[11] A. Jain, "Exploiting Long-Term Behavior for Improved Memory System Performance," Ph.D. dissertation, Austin, TX, USA, 2016.

[12] D. Kadjo *et al.*, "B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors," in *MICRO*, 2014.

[13] J. Kim *et al.*, "Path Confidence Based Lookahead Prefetching," in *MICRO*, 2016.

[14] A. Kolli *et al.*, "RDIP: Return-Address-Stack Directed Instruction Prefetching," in *MICRO*, 2013.

[15] P. Michaud, "Best-Offset Hardware Prefetching," in *HPCA*, 2016.

[16] S. H. Pugsley *et al.*, "Sandbox Prefetching: Safe Run-Time Evaluation of Aggressive Prefetchers," in *HPCA*, 2014.