TDT4260 Computer Architecture Prefetcher Lab Report Best-Offset Prefetcher

2024-03-22

Øystein B. Weibell
oystein.b.weibell@ntnu.no

Jonathan H. Kretschmer
jonathan.kretschmer@mailbox.tu-dresden.de

***Abstract*—Hardware prefetchers can improve performance of modern computer systems by mitigating impacts of memory latency. Many approaches for prefetchers exist, even for the offset prefetching family. We decide to study thoroughly Best Offset (BO) Prefetching [1] in our paper. Its complexity is quite small, also requiring little hardware overhead but it still performs well.**

This work provides an in depth study on how the BO learning algorithm works. We implement the BO Prefechter (BOP) according to the paper in our own environment. In order to find the best performing BO parameters with respect to instructions per cycle (IPC), accuracy and coverage we run a bunch of simulations over several parameters with a specific workload.

We yield a speedup of 2.44 versus no hardware prefetching on our chosen benchmark. It is also outperforming a tagged next-line prefetching scheme [2] by generating a speedup of 1.31. Our overall simulation results confirm several properties of the BOP. For example when tuning parameters for timeliness, a bit of IPC is sacrificed. They also show which trade-offs designers might be faced with, when choosing parameters like offset list, SCORE_MAX or RR table size.

I. Introduction

Hardware prefetchers play a crucial role in enhancing the performance of modern computer systems by mitigating the impact of memory latency. In some cases, even with ”large cache hierarchies [...] it is still not uncommon for many programs to spend more than half their run times stalled on memory requests” [3]. Prefetchers are designed to anticipate future data needs and proactively bring them into the cache, thus reducing the latency associated with memory accesses and the subsequent stalls.

The history of prefetchers goes back into the last several decades from simple next-line prefetching over to strided prefeching [2] and many more proposals. We focus our search on the last decade though to cover nearly state-of-the-art approaches.

Introduced by Michaud [1], the Best-Offset (BO) prefetcher is a notable example of offset prefetching. They won with their submission [4] the Second Data Prefetching Championship (DPC2) in 2015. It tackles the challenge of determining the best prefetch offset dynamically. The BO prefetcher employs a learning algorithm to assess different offsets, considering recent memory access history. This adaptability ensures that the prefetcher can adjust to changing program behaviours and varying latencies in the memory hierarchy.

In comparison to the more complex Multi-Lookahead Offset Prefetching (MLOP) [5] introduced by Shakerinava, M et al. at the Third Data Prefetching Championship (DPC3), which we also considered implementing, the BO only chooses a single ”best offset” after each evaluation period. This requires less complex hardware and makes it more feasible to implement and evaluate in the gem5 simulator, since it is a bit leaner and more straight forward. Thus, it seemed like a good choice when we consider the potential learning outcome and probability of making it functional.

This paper starts by introducing some core concepts and pitfalls when it comes to prefetchers and their implementation. It then moves on to a class of prefetchers known as Offset Prefetching, including a specific example, the Best-Offset (BO) prefetcher [1]. Finally, a modified version of the BO is presented. The section *Methodology* goes into the experimental environment and ensures that the *Results* we obtain can be replicated by others. The *Discussion* elaborates on potential experimental fallacies and dead-ends we met along the road. Finally, similar prefetchers and experiments are presented in *Related works*, to put this paper in a broader research context.

II. Background

A. Memory access patterns

Various memory access patterns influence the effectiveness of prefetching mechanisms. These patterns include, but are not limited to, sequential access, strided access, and interleaved access.

- **Sequential Access**: In this pattern, memory addresses are accessed in a continuous sequence (0, 1, 2, 3, ...). A next-line prefetcher, for example, can effectively prefetch the next line in the sequence, ensuring high coverage and accuracy.
- **Strided Access**: A strided access pattern involves accessing memory with a constant stride between addresses (0, 2, 4, 6, ...). Traditional next-line prefetchers may not

be as effective in this scenario, but an offset prefetcher with an appropriate offset value can improve coverage and accuracy.

- **Interleaved Access**: Multiple streams of memory access, as seen in interleaved patterns, present challenges for prefetching. Each stream may require a different offset to achieve optimal coverage.

B. Challenges and Latency Considerations

While prefetching can enhance coverage, accuracy, and overall system performance, challenges arise due to varying memory hierarchies and access latencies. Prefetching mechanisms must consider not only covering as much useful data as possible, but also delivering it in a timely manner to avoid performance degradation [3].

- **Latency Variability**: The latency of memory access can vary based on factors such as cache levels and whether the prefetch is a hit or a miss. Prefetchers need to adapt to this variability to provide timely data.
- **Optimal Offset**: Finding the optimal prefetch offset is a non-trivial task. It requires dynamically adjusting the offset based on the application's memory access behaviour. A fixed offset may not suffice for diverse program behaviours.
- **Cache pollution**: "A prematurely prefetched block may also displace data in the cache that is currently in use by the processor, resulting in what is known as cache pollution" [6]. "A prefetch that causes a miss in the cache that would not have occurred if prefetching was not in use is defined as cache pollution" [3].

The following two sections describe the general idea of Offset Prefetching and Best-Offset Prefetching from Michaud [1]. We follow their structure and wording, but add some illustrating examples and highlight simplifications we take for our implementation.

C. Offset Prefetching

Michaud [1] introduces offset prefetching as a generalisation of next-line prefetching.

On every core request to a line X (i.e. L2 cache miss or prefetch-hit to line X), an offset-prefetcher prefetches line $X + D$. The invokation scheme can be classified as *tagged sequential prefetching* according to VanderWiel and Lilja [2].

D is the *prefetch offset*. The case $D = 1$ corresponds to next-line prefetching. the case of arbitrary D, corresponds to offset prefetching.

The optimal prefetch offset depends on the memory access pattern of an application. Consider three different line access patterns:

1) sequential: `0,1,2,3,4,...`
2) stride=2: `0,2,4,6,8,...`
3) combined pattern: `0,3,4,6,8,9,12,...`
   - part a: stride=3: `0,3,6,9,12,...`
   - part b: stride=4: `0,4,8,12,...`

Ignoring the latency of the memory hierarchy $D = 1$ yields 100% coverage for pattern 1), $D = 2$ yields 100% coverage for pattern 2) and $D = 3 * 4 = 12$ yields 100% coverage for pattern 3).

But this naive chosen offsets do not take timeliness into account. Especially when these accesses have a high frequency and the L3 cache respective main memory have a significant latency, as it is the case on real hardware. This latency can vary between tens and hundreds of clock cycles [7, p. 20].

A *multiple* of the naively chosen offset instead may achieve timely *and* covering prefetches for real applications on real systems.

A very resource efficient algorithm to determine such offsets is given by Michaud's Best-Offset Prefetcher [1].

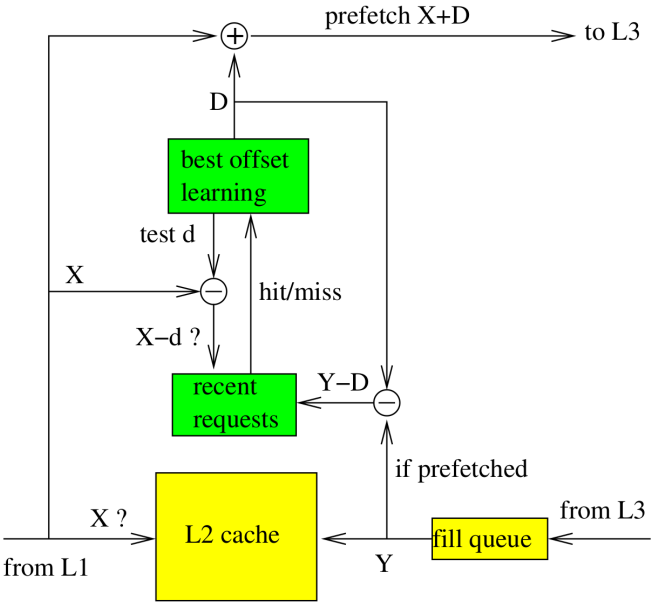


Fig. 1: Schematic view of a BO prefetcher. [1]

III. Best-Offset (BO) Prefetcher

A BO prefetcher is, above all, an offset prefetcher. It prefetches with the *current* prefetch offset D. So on every L2 miss or prefetch hit on line X, it prefetches line $X + D$.

A schematic view of a BO is given in Figure 1.

A. Best-offset learning

To determine a best offset that may replace the *current* prefetch offset D, the best-offset learning algorithm is employed (Section 4.1 [1]):

"[It] tries to find the best prefetch offset by testing several different offsets. An offset d is potentially a good prefetch offset if, when line X is accessed, there was in the recent past a previous access for line $X-d$. However, the fact that $X-d$ was recently accessed is not sufficient for guaranteeing that line X would have been prefetched in time. We want prefetches to be timely whenever possible. I.e., for d to be a good prefetch offset for line X, line $X-d$ must have been accessed recently, but not too recently. Ideally, the time between the accesses to lines $X - d$ and X should be greater than the latency for completing a prefetch request.

B. RR table

[Michaud's] solution is to record in a *recent requests* (RR) table the *base address* of prefetch requests that have been **completed**" [1].

The base address is defined as $Y - D$, where Y is a prefetched line that is ready to be placed in L2 cache. Vice versa you can think i.e. of a L2 prefetched hit on line X' 20 clock cycles ago. Then our overall offset prefetcher issues an prefetch for $X' + D$. Due to L3 latency, this prefetched line $X' + D$ arrives at first after 20 clock cycles in the fill queue from L3. Then $Y = X' + D$ and the prefeched-bit of that inserted line is set. Now we know that $Y - D$ must have been a useful for L1 some clock cycles ago (recently). That's why this base address $X' = Y - D$ is inserted into the recent requests (RR) table.

This late insert implicitly holds information about the L3 latency.

If a line $X - d$ is in the RR table, it implies, that d would have been an offset that would have prefetched line X in a timely manner.

C. Offset list

"Besides the RR table, the BO prefetcher features an *offset list* and a *score* table. The score table associates a score with every offset in the offset list. The score value is between 0 and SCOREMAX [...].

The prefetch offset is updated at the end of every *learning phase*. A learning phase consists of several *rounds*. At the start of a learning phase, all the scores are reset to 0. On every eligible L2 read access (miss or prefetched hit), we test an offset d_i from the list. If $X - d_i$ hits in the RR table, the score of offset d_i is incremented. During a round, each offset in the list is tested once: we test d_1 on the first access in the round, d_2 on the next access, then d_3, and so on. When all the offsets in the list have been tested, the current round is finished, and a new round begins from offset d_1 again." [1]

In the appended section XI we present two examples, illustrating how the best learning algorithm is scoring good offsets higher.

D. The end of a learning phase

"The current learning phase finishes at the end of a round when either of the two following events happens first: one of the scores equals SCOREMAX, or the number of rounds equals ROUNDMAX (a fixed parameter). When the learning phase is finished, we search the best offset, i.e., the one with the highest score3 . This offset becomes the new prefetch offset, and a new learning phase starts." [1]

E. Prefetch throttling

Prefetching with a fixed offset on very irregular workloads might not be efficient, with respect to prefetch accuracy. This can result in replacement of useful cache lines.

Therefore the BO prefetcher employes a metric for switching off prefetching. When no score is above a threshold BADSCORE after a finished learning phase, prefetching is switched off.

But the BO learning proceeds. Figure 1 illustrates the case for prefetching enabled. But now the tag itself (Y-0) filled into L2 is always added to RR table - not just if it is a prefetched fill.

IV. Implementation: Our Best-Offset Prefetcher

A. Configuration parameters

We declare several configuration parameters: `SCORE_MAX`, `ROUND_MAX` and `BAD_SCORE`. In addition we define `NUMBER_OF_OFFSETS` and `RR_SIZE`, to be able to perform simulations with a subset of the offset-list and different RR table sizes.

B. Offset list

The authors of the paper [1] use a list of prefetch offsets between 1 and 256, where $d \in \{2^i * 3^j * 5^k\}$. The BO prefetcher is not prefetching accross page boundaries. Our system has 64B cache lines and we assume 4KiB pages. But according to the paper the BO prefetcher does not prefetch across page boundaries. Therefore we limit the offsets to be smaller than 64. Then our offset list consists of 26 entries: 1,2,3,4, 5,6,8,9, 10,12,15,16, 18,20,24,25, 27,30,32,36, 40,45,48,50, 54,60.

C. Internal State

First of all the internal state consists of the currently used `bestOffset` for prefetching.

The paper states that several implementations of the *RR table* are possible. For the sake of simplicity we decided against modelling a direct-mapped like cache, accessed through a hash function, as proposed by the paper. Instead we imitate a simple ring buffer, by using a double-ended queue (`std::deque` of `int`), to implement the recent requests list.

Furthermore a *flag* that is indicating whether prefetching is *enabled* and counters for the `round` and the `subround`. The latter one is corresponding to i pointing to the offset d_i to be tested on one eligible L2 access and is incremented on every access. The `subround` counter is beeing reset when it is equal to `NUMBER_OF_OFFSETS`. The `round` counter is reset when the end of a learning phase is reached. (See section Best Offset Learning).

D. Functions

Our implementation derives from the gem5 `Queued : Base` prefetcher.

In order to model the tagged prefetching behaviour and making sure the method `calculatePrefetch` is beeing invoked only on L2 read accesses that are misses or prefetched hits, we set the `Base` prefetcher parameters `on_miss`, `on_read`, `on_data`, `on_inst` and `prefetch_on_pf_hit`) to *true*, and `on_write` and `prefetch_on_access` to *false*.

We also implement methods `notifyFill` and `notifyPrefetchFill` that are beeing invoked when a cache line from L3 is filled into L2 cache, the latter one

only when the line has been prefetched. This is required by the scheme (see figure 1).

The implementation of the event listener for `notifyPrefetchFill` has been provided by our teachers. (https://github.com/davidmetz/gem5-tdt4260/tree/student)

E. Ressource usage

The BO prefetcher generally has a small hardware overhead [1]. In addition to the (very small) offset prefetching overhead, just a bit of bookkeeping for recent requests and scores is required.

However for the sake of a readable implementation we use software data structures that were not meaningfully directly implemented in hardware, for example the `deque` and the `vector`. Instead the RR table might be represented for example by a little cache providing hits or misses and indexed by a hash function [1]. This reduces memory usage significantly in contrast to saving the pure full addresses. Also the offset list might be hard-coded in read only memory (ROM).

F. Environment

We made only few adjustments to the provided baseline architecture configuration of gem5. It has been provided to us by our teachers.[1] It comes with a single out-of-order O3 CPU. The system is modelled according to parameters published by Intel in their "Register file prefetching" paper [8] to match an Intel Ice Lake CPU. The L1 data cache size is 48KiB, associativity 12; L1 instruction cache size is 32KiB; L2 cache size is 1280KiB, associativity 20; and L3 cache size is 3MiB, associativity 12. The cache line size is 64B.

We disable hardware prefetching except for the L2 cache, where we employ our Best Offset prefetcher.

V. Methodology

We want to find the best performing configuration of BO parameters on our system for a given workload.

We measure performance by instructions per cycle (IPC), accuracy[2] and coverage.[3]. All these performance parameters are being generated automatically on a run of the provided `run_prefetcher.py` script.

Due to time constraints we limit our search on three changed parameters and only one benchmark.

We utilise the provided SPEC2017 gcc_s benchmark [9]. The prefetcher is employed and the statistics are being generated after some warm-up of the benchmark. The number of instructions executed by the simulator is limited to 10 million.

In accordance to [1, table 2] we keep following parameters fixed: `ROUND_MAX` $= 100$ and `BAD_SCORE` $= 1$. But we change simulate over differing RR table size, offsets and `SOCRE_MAX`:

- `RR_SIZE` $= \{16, 32, 64, 128, 256\}$
- a subset (just the first 16 entries) and the full offset list[4]
- `SCORE_MAX` $= \{1, 2, \ldots, 100\}$

This gives a total of $2 * 5 * 100 = 1000$ simulations.

[1] available under https://github.com/davidmetz/gem5-tdt4260/tree/student

[2] Measures the number of useful prefetches issued by the prefetcher.

[3] How many of the potential candidates for prefetches were actually identified by the prefetcher?

[4] `NUMBER_OF_OFFSETS` $= \{16, 26\}$

	No Prefetcher	Tagged Next-Line	BO Maximise IPC	BO Maximise Accuracy
IPC	0.236531	0.441386	**0.577577**	0.522890
Speedup	1.000000	1.866081	**2.441866**	2.210662
Accuracy	n/a	0.366883	0.899859	**0.920615**
Coverage	n/a	0.901658	0.715169	0.670101
`RR_SIZE`	n/a	n/a	16	64
`SCORE_MAX`	n/a	n/a	7	55
Offsets	n/a	n/a	all 26	just 16

TABLE I: Comparison of the best performing BO prefetcher configurations against a reference with no prefetching and simple next-line prefetching. Benchmark: gcc_s, `ROUND_MAX` $=$ 100 and `BAD_SCORE` $= 1$

VI. Results

Table I presents the best performing configurations found with the simulations. Additionally we make some interesting observations.

Coverage and IPC are strongly correlated as you can see in figure 2. Therefore we omit a column showing the results for maximising coverage as they are very similar to those for maximising IPC.

Optimising for IPC, accuracy or coverage may result in quite different BO parameters. BO can easily outperform next-line with respect to IPC and accuracy but not with respect to coverage.

When maximising accuracy we encounter a more worse IPC. A bigger RR table and the smaller offset subset are beneficial for accuracy.

When maximising IPC we encounter $1.2\,\%$ late prefetches in contrast to when maximising for accuracy, where we encounter just $0.6\,\%$ late prefetches. This confirms Michaud's statement that striving for timeliness may not always be optimal [4, section 2.3].

Generally increasing `SCORE_MAX` is beneficial for performance except for either small RR tables sizes (see figure 2a) or less number of offsets (see figure 3d, blue line).

We also saw that using a smaller set of offsets is very beneficial. See figure 3 where for a sufficiently large RR table size and the small offset set (≤ 25), any chosen `SCORE_MAX` is outperforming the full set of offsets (≤ 60). This may be caused firstly by smaller offsets generally being beneficial for performance (ignoring timeliness) and secondly by the relatively high `ROUND_MAX` parameter. With less number of offsets the prefetcher might become more flexible on changing request patterns because learning phases become shorter.

It turned out that performance for RR tables larger than 64 entries is stays quite similar or is even degrading. We omit related figures. The degrading effect may come from the BO algorithm wrongly scoring offsets that only satisfy *out-dated*

request patterns. Therefore the RR table should be choosen not too large [5, section 3.1].

VII. Discussion

The used environment tires to model a nowadays actually used CPU type. Also the simulations aim to cover a huge parameter space allowing to find a well performing configuration for the given experimental setup quite confidentially.

On the other hand the performance of a specific BO prefetcher parameter configuration is substantially depending on the workload. The gcc_s workload might not fully have a request patterns where offset prefetching might be meaning ful at all. Therefore just optimising for one benchmark is by far not enough.

Our implementation of the RR table is allowing duplicate entries. This might degrade performance on specific workloads dramatically, because the RR queue/`std::deque` might be spammed with the same address and the amount of information contained in the RR table would decrease. A sufficiently large queue might decrease the impact of duplicate fills. In a new iteration of the implementation, a set-like implementation (not allowing multiple duplicate entries) or maybe even a direct-mapped cache like implementation should be considered, as proposed by [1, section 4.4].

A. Resource Usage

Allowing duplicate entries in the RR table and therefore maybe increasing the number of RR table entries is a waste prefetcher of memory.

With BO prefetching we encounter a better accuracy compared to next-line prefetching. This implies that more prefetched lines were actually referenced by the CPU before replaced in the cache. Therefore cache pollution and memory bandwidth pollution is lower, resulting in less energy usage and better IPC.

VIII. Related Work

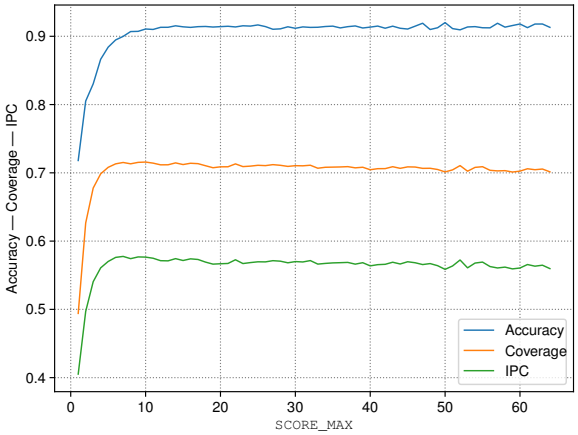
In the BOP papers they refer amongst others to sandbox prefetching (SP) by Pugsley et al. [10] as one of the first offset prefetchers. Sandbox prefetching is testing and performing prefetches with several different offsets but not taking timeliness into account [4, section 4].

Michaud's BOP then trys to take timeliness into account and is the winner of DPC2.

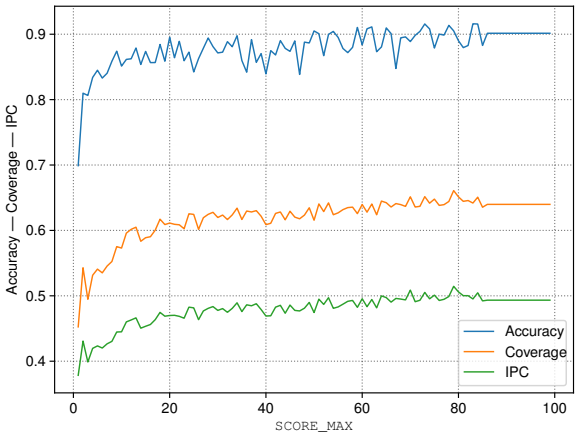
Later at DPC3 Shakerinava et al. propose Multi-Lookahead Offset Prefetching (MLOP) [5] as an improved offset prefetcher that is trying to take both coverage and timeliness into account. This is achieved by considering multiple *prefetching lookaheads* during the evaluation of prefetch offsets.

IX. Conclusion

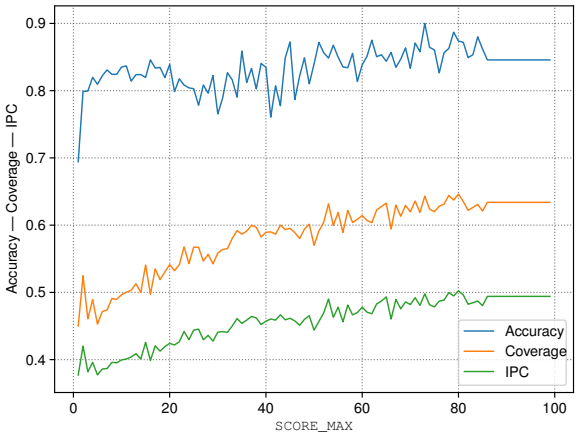
In this paper we have studied the behaviour of the BOP with different parameters. We could learn a lot about different quite modern prefetching techniques with little hardware overhead.



(a) `RR_SIZE` = 16, NOTE: This specific simulation terminated at score max 65. We could not figure out why, yet.

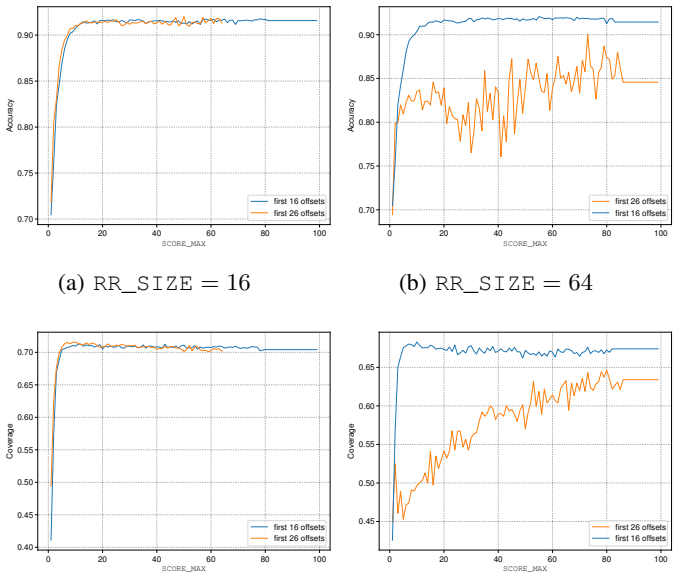


(b) `RR_SIZE` = 32



(c) `RR_SIZE` = 64

Fig. 2: Accuracy, Coverage and IPC plotted over varying `SCORE_MAX`. Full 26-entries offset list is being used. When `SCORE_MAX` is approaching `ROUND_MAX` some sort of saturation is happening.



(a) RR_SIZE = 16 (b) RR_SIZE = 64

(c) RR_SIZE = 16 (d) RR_SIZE = 64

Fig. 3: Accuracy (upper part) and Coverage (lower part) plotted over varying SCORE_MAX. Blue: 16 offsets; Orange: 26 offsets. Especially for bigger RR table size, greater offsets seem to worsen the prefetching performance.

Our BOP out-performs simple next-line prefetching on the gcc_s benchmark with respect to IPC and accuracy. We also provide ideas on how to tune parameters.

Future work should utilise more different benchmarks to model a more realistic workload. Also the implementation of the RR table should be refactored to fix problems that may arise from duplicated entries. The impact of varying the offset list and their interaction with variying RR table size might also be studied more in depth.

References

[1] P. Michaud, "Best-offset hardware prefetching," *International Symposium on High-Performance Computer Architecture*, 2016.

[2] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, p. 174–199, jun 2000.

[3] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, p. 174–199, jun 2000.

[4] P. Michaud, "A best-offset prefetcher," *DPC2*, 2015.

[5] M. Shakerinava *et al.*, "Multi-lookahead offset prefetching," *International Symposium on Computer Architecture (ISCA)*, 2019.

[6] J. Casmira and D. R. Kaeli, "Modeling cache pollution," *In Proceedings of the Second IASTED Conference on Modeling and Simulation*, pp. 123–126, 1995.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2017.

[8] S. Shukla, S. Bandishte, J. Gaur, and S. Subramoney, "Register file prefetching," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 410–423, Association for Computing Machinery, 2022.

[9] R. Stallman *et al.*, "602.gcc_s, spec cpu®2017 benchmark description," 2017. https://www.spec.org/cpu2017/Docs/benchmarks/602.gcc_s.html, last updated: 2020-09-23.

[10] S. H. Pugsley, Z. A. Chishti, C. Wilkerson, P. fei Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 626–637, 2014.

X. Use of AI for this work

Which parts contain contributions of AI? A code editor copilot has been used while writing code for the prefetcher implementation, especially when struggling with C/C++ details and for scripting python code generating diagrams from the simulation data.

Which tools have been used? VS Code plugin Codeium.

XI. Appendix: Illustrating Examples for the BO learning phase

We present two example executions of the learning phase. These show the ability of the algorithm to find good offsets in presence of high latencies of the L3 cache on the one hand and interleaved strided access patters on the other hand.

First consider a 'large' L3 cache latency of 5 clock cycles and a simple access pattern `0,2,4,...` with stride 2. The current prefetch offset is $D = 2$ and the offset list contains $d_i = 6, 12, 18$. Figure 4 shows the first 7 rounds of the learning phase. The score for offset 12 is higher than for 6 and 18. Offset 6 is too small to provide timely prefetches and 18 also not favoured at the beginning as Y - d results in base addresses that have not been requested – 'it goes too far back in time'. Here the algorithm find the offset that provides timely prefetches.

D =	2																		
		cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		req. line X from L1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
		prefetch = X+D	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36
		lines Ys from L3 fill queue (only prefetches)	L3-latency = 5 cycles					2	4	6	8	10	12	14	16	18	20	22	24	26
		RR table insertion						**0**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**
		round		1			2			3			4			5			6	
		d	6	12	18	6	12	18	6	12	18	6	12	18	6	12	18	6	12	18
		X-d	**-6**	**-10**	**-14**	**0**	**-4**	**-8**	**6**	**2**	**-2**	**12**	**8**	**4**	**18**	**14**	**10**	**24**	**20**	**16**
offset	**score**																			
6	0	increment offset	0			0			0			0			0			0		
12	4	score by		0			0			1			1			1			1	
18	3				0			0			0			1			1			1

Fig. 4: Example prefetcher execution for a strided pattern and large latency. BO learning favours timely offsets.

Secondly consider a 'negligible' L3 cache latency of 1 clock cycle, but a more complex access pattern of two interleaved patterns with strides 3 and 4. The current prefetch offset is $D = 2$ again. The offset list contains $d_i = 1, 2, 3, 4, 12$. Figure 5 shows the first 7 rounds of the learning phase. The score for the offsets 1 and 2 are smaller than for 3 and 4. Which highlights that in at least some rounds these are considered to be a good prefetch offset. But after some time the prefetch offset 12 is always considered as a good prefetch offset, because it is a multiple of 3 and 4 – the strides. Here the algorithm finds the offset that is a multiple of all strides.

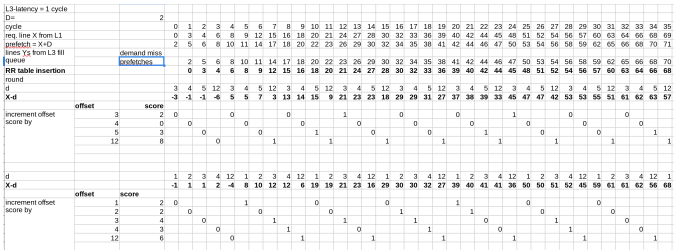


Fig. 5: Example prefetcher execution for interleaved access patterns. BO learning favours multiple of both strides. (NOTE: only execution for offsets 1,2,3,4,12 (lower part) is relevant)