A Best-Offset Prefetcher

Pierre Michaud

A Best-Offset Prefetcher

Pierre Michaud
Inria
pierre.michaud@inria.fr

The Best-Offset (BO) prefetcher submitted to the DPC2 contest prefetches one line into the level-two (L2) cache on every cache miss or hit on a prefetched line. The prefetch line address is generated by adding an offset to the demand access address. The BO prefetcher tries to find automatically an offset value that yields timely prefetches with the highest possible coverage and accuracy. It evaluates an offset value by maintaining a table of recent requests addresses and by searching these addresses to determine whether the line currently requested would have been prefetched in time with that offset.

The paper is organized as follows. Section 1 presents offset prefetching. Section 2 describes the proposed BO prefetcher. Section 3 describes a prefetch throttling mechanism (somewhat specific to DPC2) for dealing with situations where prefetch hurts performance. Section 4 mentions related prefetchers. Finally, Section 5 presents a few simulation results.

1 Offset prefetching

The BO prefetcher is an offset prefetcher, like the Sandbox[1] prefetcher [11]. Offset prefetching superficially resembles stride prefetching and stream prefetching, but is a more aggressive prefetching method.

Offset prefetching is a generalization of next-line prefetching [11]. A next-line L2 prefetcher works as follows: when a cache line of address X is requested by the level-1 (L1) cache, a next-line prefetcher prefetches the line of address X+1 into the L2 cache[2]. The accuracy of next-line prefetching can be improved with a *prefetch bit* stored along with each L2 line. The prefetch bit is set if the line has been prefetched but not yet requested by the L1 cache. It is reset if and when the line is requested by the L1 cache. A prefetch request for line X+1 is issued only if line X misses in the L2 cache, or if the prefetch bit of line X is set (prefetched hit) [15].

Offset prefetching generalizes next-line prefetching: for every line X requested by the L1 cache which is a L2 miss or a prefetched hit, prefetch line X+O into the L2 cache, where O is a non-null offset value [11].

The optimal offset may not be the same for all programs. It may also vary as the program behavior changes. We present in Section 2 a new method for finding the best offset automatically.

Offset prefetching exploits a form of spatio-temporal address correlation. The remainder of this section provides a few examples of such correlations. The following examples assume 64 byte lines, as in the DPC2 simulator. For convenience, lines requested by the L1 cache in a memory region are represented with a bit vector, adjacent bits representing adjacent lines. The bit value tells whether the line is accessed ("1") or not ("0"). We ignore the impact of page boundaries and consider only the steady state on long access streams.

1.1 Example 1: sequential stream

Consider the following sequential stream:

```
11111111111111111...
```

That is, the lines accessed are X, X+1, X+2, and so on. A next-line prefetcher yields 100% prefetch coverage and accuracy on this example. However, issuing a prefetch for X+1 just after the access to X might be too late to cover the full latency of fetching X+1 from the last-level cache (LLC) or from memory, leading to a *late prefetch*. Late prefetches may accelerate the execution[3], but not as much as timely prefetches. An offset prefetcher yields 100% prefetch coverage and accuracy on sequential streams, like a next-line prefetcher, but can deliver timely prefetches if the offset is large enough.

Another factor that may degrade prefetch coverage is *scrambling*, i.e., the fact that the chronological order of memory accesses may not match the program order exactly [6]. Some scrambling at the L2 cache may be caused by out-of-order scheduling of instructions and/or L1 miss requests. Some scrambling at the L2 may also happen if there is an L1 prefetcher (which is not the case in the DPC2

[1]The BO prefetcher is nevertheless different from the Sandbox prefetcher, there is no sandbox in it (cf. Section 4).

[2]The address of a line is the address of the first byte in that line divided by the line size in bytes.

[3]Thanks to the L2 MSHR. The L2 MSHR is a fully associative structure holding all the pending L2 miss and prefetch requests. If there is already a pending prefetch request for a missing L2 line, the miss request is dropped and the prefetch request is promoted to demand request [16].

simulator). If the offset is large enough, an offset prefetcher is more robust against scrambling than a next-line prefetcher.

1.2 Example 2: strided stream

Consider a load instruction accessing an array with a constant stride of +96 bytes. With 64-byte cache lines, the bit vector of accessed lines is

```
110110110110110110...
```

A next-line prefetcher can only prefetch every other accessed line. A delta correlation prefetcher observing L1 miss requests (such as AC/DC [9]) would work perfectly here, as the sequence of line strides is periodic (1,2,1,2,...). But a simple offset prefetcher with a multiple of 3 as offset yields 100% coverage and accuracy on this example.

Offset prefetching can (in theory) deliver 100% coverage and accuracy on any periodic sequence of line strides, by setting the offset equal to the sum of the strides in a period, or equal to a multiple of that number.

1.3 Example 3: interleaved streams

Consider two interleaved streams S1 and S2 accessing different memory regions and having different behaviors:

```
S1: 101010101010101010...
S2: 110110110110110110...
```

Stream S1 alone can be prefetched perfectly with a multiple of 2 as offset. Stream S2 alone can be prefetched perfectly with a multiple of 3 as offset. Both streams can be prefetched perfectly with a multiple of 6 as offset.

2 The BO prefetcher

A full-fledged offset prefetcher should have a mechanism for finding automatically the best offset value for each application. Otherwise, if we had to choose a single fixed offset for all applications, the best offset value would probably be 1. That is, the resulting prefetcher would be a next-line prefetcher. The offset could be set by using hints from the programmer or, as we propose here, with a hardware solution.

We store in a Recent Requests (RR) table the *base* address of each recent prefetch, that is, the address of the demand access that generated the prefetch. Precisely, when a prefetched line Y is entered into the L2 cache (even if that line was demand-accessed after the prefetch request was issued), we write address $Y - O$ into the RR table, where O is the *prefetch offset*, i.e., the offset currently used for prefetching.

We consider a list of possible offsets. This list is fixed at design time. We associate a *score* with every offset in the list. On every L2 miss or prefetched hit for a line address X, we test the n^{th} offset O_n from the list by searching if the line address $X - O_n$ is in the RR table. If address $X - O_n$ is in the RR table, this means that line X would likely have been prefetched successfully with offset O_n, and the score for offset O_n is incremented. The next L2 access tests offset O_{n+1} and so on. A *round* corresponds to a number of accesses equal to the offset list size. Immediately after a round is finished (i.e., all the offsets have been tested once), n is reset and a new round starts. The prefetcher counts the number of rounds. A learning *phase* finishes at the end of a round when either of the two following events happens first: one of the scores equals SCORE_MAX, or the number of rounds equals ROUND_MAX. In our DPC2 submission, we set SCORE_MAX=31 and ROUND_MAX=100. At the end of a learning phase, we search the *best* offset, i.e., the offset with the highest score. This best offset will be the prefetch offset during the next phase. Then, the scores and the round counter are reset, and a new phase starts. The offsets are continuously evaluated, the prefetch offset tracking the program behavior.

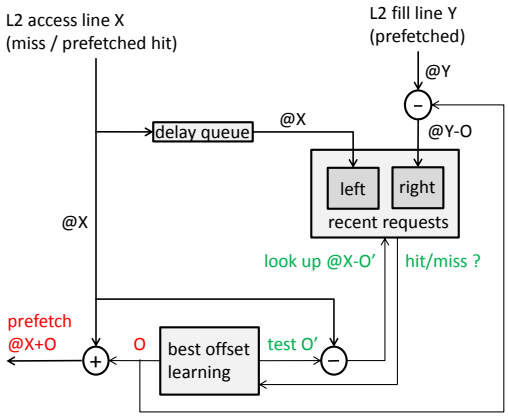

Figure 1: **The BO prefetcher**

Figure 1 shows a sketch of the proposed prefetcher. The remainder of this section provides a detailed description.

2.1 Recent Requests (RR) table

Several different implementations are possible for the RR table. We propose to implement it like the tag array of a 2-way skewed-associative cache [13]. There are two banks, indexed through different hashing functions. Unlike in a skewed-associative cache, we write the line address in both banks. It is not necessary to write the full line address, a partial tag is enough. In our DPC2 submission, each of the two banks has 64 entries, and each entry holds a 12-bit tag.

A hit in the RR table happens when there is a tag match in any of the two banks. A write and a read could happen during the same cycle. In our DPC2 submission, we assume that the RR table is dual-ported. However in a real processor, dual-porting is not necessary. The bandwidth requirement of the RR table is much less than one read and one write every clock cycle. A little buffering should solve conflicts easily.

2.2 List of offsets

Because DPC2 prefetching is limited by 4KB-page boundaries, we only consider offsets less than 64. Considering both positive and negative offsets, there are up to 126 possible offsets. However in practice, the list needs not contain all the offsets in an interval. From our experiments with the SPEC benchmarks, considering only offsets of the form $2^i \times 3^j \times 5^k$, with $i,j,k \geq 0$, seems to work well in practice. This rule of thumb is particularly useful when the page size is greater than 4KB, as there is some benefit in considering large offsets.

In our DPC2 submission, the list contains 46 offsets (23 positive, 23 negative). The offset list is hard coded in a ROM. Each score is coded on 5 bits.

2.3 Delay Queue

In an initial implementation of the BO prefetcher, both banks of the RR table were written simultaneously when a prefetched line is inserted into the L2 cache. The rationale for updating the RR table at L2 fill time is to find an offset that yields timely prefetches whenever possible. However, we later found that striving for timeliness is not always optimal. There are cases where a small offset gives late prefetches but greater coverage and accuracy.

We propose an imperfect solution to this problem[4]: a *delay queue*. Instead of writing simultaneously in both banks of the RR table at L2 fill time, we only write in the "right" bank. When a prefetch request $X+O$ is issued, the base address X is enqueued into the delay queue. The delay queue holds address X for a fixed time. After this time has elapsed, address X is dequeued and is written into the "left" bank of the RR table.

We choose the delay value between the latency of an LLC hit and that of an LLC miss. In our DPC2 submission, the delay queue has 15 entries and a delay of 60 cycles. Each entry holds 18 address bits for writing the "left" bank of the RR table, one valid bit, and a 12-bit time value for knowing when to dequeue[5], for a total of $18+1+12=31$ bits per entry.

3 Prefetch throttling

For some applications, prefetching may hurt performance, in particular by polluting the L2 and LLC or by wasting memory bandwidth. Several approaches are possible for solving this problem in hardware:

Conservative prefetching. Issue a prefetch request only when it is likely to be useful. In particular, this approach is used in stride prefetchers [1, 3, 10, 14].

Prefetch throttling. Adjust prefetch aggressiveness dynamically [2, 5, 16, 18], e.g., by monitoring certain events (prefetch accuracy, cache pollution,...), possibly disabling the prefetcher when it hurts performance [2, 9]. Prefetch throttling is good both for worst-case performance and energy consumption.

Try to make useless prefetches harmless. Request schedulers should prioritize miss requests over prefetch requests [8, 9]. To deal with cache pollution, we should either use prefetch buffers [7] or implement a prefetch-aware cache insertion policy [8, 16, 17, 12].

Conservative prefetching does not apply in our case, offset prefetching being an aggressive prefetching method. Implementing an efficient offset prefetcher requires combining prefetch throttling and prefetch-aware request schedulers and cache insertion policies. However, the DPC2 contest allows to modify neither the request schedulers nor the cache insertion policy. So we only have a prefetch throttling mechanism, described thereafter. Our throttling mechanism is somewhat specific to the DPC2 simulator.

The DPC2 prefetcher can only observe what happens at the L2 cache. In particular, there is no way to directly measure LLC pollution and memory traffic. We use two knobs for throttling prefetches: dropping prefetch requests when the MSHR occupancy exceeds a threshold[6], or turning prefetch off completely. We monitor two quantities for controlling these knobs: the best score, and the LLC access rate.

3.1 Turning prefetch off and on

At the end of a learning phase, we not only find the best offset, we also know its score. In particular, if the best score is less than SCORE_MAX, it provides an estimate for the prefetch accuracy of the best offset. If the best score is less than or equal to a fixed value BAD_SCORE, we turn prefetch off during the next phase.

While prefetch is off, best-offset learning must continue so that prefetch can be turned on if the program behavior changes and the score exceeds BAD_SCORE. Hence when prefetch is off, every line that is inserted into the L2 has its address inserted simultaneously into the "right" bank of the RR table, and every L2 miss address is pushed into the delay queue.

In our DPC2 submission, we set BAD_SCORE=1 for the 1MB LLC and BAD_SCORE=10 for the 256KB LLC.

3.2 MSHR threshold

The DPC2 contest does not allow to modify request schedulers. In particular, useless prefetches may hurt performance by wasting memory bandwidth. To mitigate this problem, we decrease the MSHR threshold when we detect

[4] This is a topic for future research.

[5] The DPC2 prefetcher can modify its state only at cycles corresponding to L2 accesses, which is an artificial constraint. In a real processor, time values are not necessary. A delay of 60 cycles can be implemented just by moving a token in a 15-entry circular shift register every 4 cycles. Only the queue entry which has the token can be read or written.

[6] as done in some example prefetchers provided with the DPC2 simulation infrastructure.

that memory bandwidth is close to saturation. Because the DPC2 prefetcher cannot monitor bandwidth usage directly, we use an approximate method: we estimate the LLC access rate, and we use it as a proxy for memory bandwidth usage.

The LLC access rate is estimated as follows. We compute the time difference Δt in cycles between the current LLC access (L2 miss or prefetch request) and the previous one, and we use two up-down saturating counters: a *gauge* counter, and a *rate* counter. The rate counter gives an estimate of the average time between consecutive LLC accesses. We update the gauge on every LLC access:

$$\text{gauge} \leftarrow \text{gauge} + \Delta t - \text{rate}$$

If the gauge reaches its upper limit we increment the rate, if the gauge reaches its lower limit we decrement the rate, otherwise we leave the rate unchanged.

We set the MSHR threshold depending on the rate counter value and on the best score. If the best score is above a fixed value LOW_SCORE, or if the LLC rate is greater than a fixed value 2×BW, we set the MSHR threshold to 12 (i.e., the prefetch request is dropped if the current MSHR occupancy is ≥ 12). Otherwise, if the LLC rate is less than BW, we set the MSHR threshold to 2. Otherwise (LLC rate between BW and 2×BW), we set the MSHR threshold to $2+10\times(\text{rate}-\text{BW})/\text{BW}$.

In our DPC submission, LOW_SCORE=20 and BW=64 for the low bandwidth configuration and BW=16 for the other configurations.

4 Related prefetchers

To the best of our knowledge, the prefetchers most similar to the BO prefetcher are the ROT (right-on-time) prefetcher [4] and the Sandbox prefetcher [11].

The ROT prefetcher is a stream prefetcher where each stream may have a different stride [4]. In order to achieve prefetch timeliness, the prefetch distance is increased automatically upon detecting late prefetches. The ROT prefetcher detects streams by associating a score with every stride in a list of stride candidates. The method for updating scores is somewhat similar to the method we use in the BO prefetcher to find the best offset. The ROT prefetcher discards strides with a low score.

The Sandbox prefetcher is the only other offset prefetcher that we are aware of [11]. To test an offset, the Sandbox prefetcher does fake prefetches with that offset by setting corresponding bits in a Bloom filter. If a subsequent demand request hits on these bits, the fake prefetch is deemed successful. Prefetch timeliness is not taken into account, unlike in the BO prefetcher. The Sandbox prefetcher can issue prefetches for several different offsets simultaneously if their accuracy is above a threshold, while the BO prefetcher uses a single prefetch offset for a period of time.

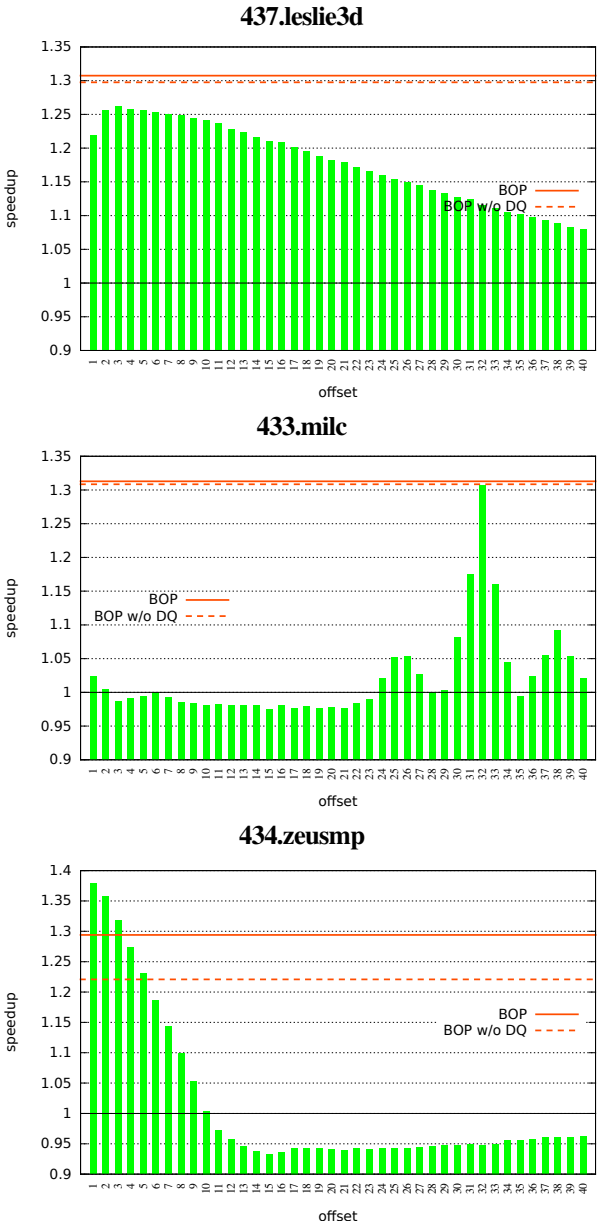

Figure 2: **Fixed offset vs. adaptive offset (BOP)**

5 Some simulation results

Figure 2 shows a few experimental results obtained with the DPC2 simulator on some prefetch-friendly execution samples from the SPEC CPU2006 benchmarks[7], assuming configuration 1 of the DPC2 contest (1MB LLC, 12.8 GB/s memory bandwidth). We compare the best-offset prefetcher (BOP) described in the previous sections with fixed-offset prefetchers,

[7] A sample is not necessarily representative of the whole benchmark.

where we vary the fixed offset[8] from +1 to +40. The y-axis gives the speedup over a baseline with prefetching disabled.

The first example in Figure 2 (437.leslie3d) is typical of sequential streams. Here, a fixed offset of +1 is suboptimal, giving late prefetches. The optimal fixed offset is +3. For offsets beyond 3, the speedup decreases gently, which is an effect of 4KB-page boundaries. Here, the BO prefetcher outperforms the best fixed offset, being able to exploit varying program behaviors.

The second example in Figure 2 (433.milc) corresponds to streams with a non-unit stride, with speedup peaking at a fixed offset of 32. Here the BO prefetcher produces the same speedup as the best fixed offset.

The last example in Figure 2 (434.zeusmp) illustrates the impact of the delay queue. The dashed horizontal line represents the speedup of the BO prefetcher but without the delay queue, i.e., both banks of the RR table are written when a prefetched line is inserted into the L2. While on the first two examples the delay queue makes little difference, it increases performance on 434.zeusmp. Here, the optimal fixed offset is +1. However, the speedup falls rapidly as the fixed offset grows. The descent is steeper than what a long sequential stream would produce (cf. 437.leslie3d). This steep descent is due to very short sequential streams [5]. Without the delay queue, the BO prefetcher selects an offset large enough for prefetches to be timely, which substantially degrades coverage and accuracy. The delay queue, with its delay less than the L3 miss latency, allows the BO prefetcher to select a smaller offset. However, this example also shows that the delay queue is an imperfect solution, the BO prefetcher being outperformed by next-line prefetching.

prefetch bits (1 bit per L2 line)	2048
RR table (2x64 entries, 12-bit tags)	1536
delay queue (15 x 31 bits, 2 pointers)	473
46 scores of 5 bits	230
round counter	7
offset list iterator	6
prefetch offset	7
score of the prefetch offset	5
best offset	7
highest score	5
MSHR threshold	4
gauge counter	13
rate counter	8
previous LLC access cycle	12
Total	**4361**

Table 1: **BO prefetcher state (number of bits)**

Acknowledgments

This work was partially supported by the European Research Council Advanced Grant DAL No 267175.

References

[1] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5), May 1995.

[2] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP*, 1993.

[3] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *MICRO*, 1992.

[4] E. Hagersten. *Toward scalable cache only memory architectures*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 1992.

[5] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO*, 2006.

[6] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 13, January 2011.

[7] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *ISCA*, 1990.

[8] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA*, 2001.

[9] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: an adaptive data cache prefetcher. In *PACT*, 2004.

[10] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA*, 1994.

[11] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: safe run-time evaluation of aggressive prefetchers. In *HPCA*, 2014.

[12] V. Seshadri, S. Yekdar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization*, 11(4), January 2015.

[13] A. Seznec. A case for two-way skewed-associative caches. In *ISCA*, 1993.

[14] I. Sklenar. Prefetch unit for vector operations on scalar computers. *Computer Architecture News*, 20(4), September 1992.

[15] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3), 1982.

[16] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[17] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer. PACMan: prefetch-aware cache management for high performance caching. In *MICRO*, 2011.

[18] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP*, 2003.

[8] Among our execution samples, a single one benefits from negative offsets.