BL∪E: A Timely, IP-based Data Prefetcher

Alberto Ros
Computer Engineering Department
University of Murcia
Murcia, Spain
aros@ditec.um.es

***Abstract*—High-performance prefetchers require not only predicting the future cache lines that will be requested but also when they will be requested. Timeliness is therefore an essential property for getting the maximum performance from a prefetcher. Bringing the cache line too early to cache can decrease the coverage of the prefetcher when such cache line is evicted before is requested. On the other hand, prefetching the data too late can lead to late prefetchers, and thus, sub-optimal performance.**

This paper presents BL∪E, a data prefetcher that predicts the prefetched cache lines based on timeliness. The prefetcher accounts for the time required to fetch a cache line and issues the prefetch request early enough, such that when it is accessed it will already be stored in cache. For each instruction pointer (or group of them) BL∪E i) correlates in a timely way the cache lines that have been requested and ii) infers their timely delta when the cache lines have not been accessed yet.

I. Introduction and Motivation

Timely is an essential property for high-performance prefetchers. Traditionally, timely has been achieved by adapting the degree of prefetch, i.e., how many prefetches are issued. For example, when considering accesses to contiguous cache lines, a next-line prefetcher [1] will predict correctly all cache lines that will be accessed, but probably they will be accessed before arriving to cache. Timely prefetchers are achieved in this case by increasing the degree of prefetch to d and requesting the next d cache lines on an access. The degree can be adapted at run-time to achieve a good balance between timeliness and accuracy [2].

Recently, Michaud proposed the best-offset prefetcher (BOP) [3], which was the best performing prefetching technique in the 2nd Data Prefetching Championship. The key difference of BOP with respect to the previous proposal is achieving timely prefetches with a prefetch degree of one, that is, issuing a single prefetch per cache access. BOP finds the best delta for the accesses performed by an application, and applies it to the next accesses.

Our prefetching mechanism, BL∪E, is based on the observation that timeliness, and therefore the best delta, varies from instruction to instruction and even from access to access, and a global delta results in sub-optimal performance. As a consequence, our prefetching mechanism finds the most suitable delta per address and instruction.

BL∪E is a prefetcher comprised of three independent but complementary prefetchers. The main prefetcher is an extension of Berti [4] optimized to have a contiguous view of the non-contiguous physical address space. This prefetcher is able to predict a delta based on the instruction pointer or the page accessed, even if a cache line or page has not been accessed before. The second prefetcher, based on the concept of Entangling [5], is the most precise and the less likely to be triggered, as it requires the cache line to be previously accessed in order to trigger a prefetch for that cache line. Finally, the last prefetcher is a next-line prefetcher with a degree of two, used when the previous prefetchers have not been able to trigger two prefetchers. The next sections describe these prefetchers.

II. Berti prefetcher

The main goal of Berti is to collect two pieces of information about the pages that have been accessed, both of them independent of the order in which the accesses took place: the cache lines (actually page offsets) accessed within each memory page and the delta that provides more timely prefetches for each memory page, namely **be**st-**r**equest-**ti**me delta, or for short, Berti delta. This information is then leveraged to predict which cache lines are prefetched in a timely manner.

Let us consider the accesses to a memory page in *429.mcf-217B* (SPEC CPU 2006). The information collected by the Berti prefetcher is shown in TABLE I. The accessed cache lines are represented in a bit vector, where 1 indicates that the cache line with a page offset equal to its position in the vector has been accessed. The Berti delta for this page is -6 which means that the timely cache line corresponds to six offsets before the current access.

TABLE I: Example found in *429.mcf-217B*

Cache lines accessed [0..63]	Berti
...**11**0**11**0110110	-6

In bold are represented the accesses that are prefetched in a timely manner according to the indicated Berti value. The remaining of the accesses for that page (...) will be also prefetched in a timely manner. However, in red are represented the first accesses to this page, that cannot be prefetched with a previous access to that page. The order to achieve timely prefetches for all the accesses to a page, we propose the **lin**k **ne**xt **a**ddress (Linnea) optimization, which correlates physical pages and offers to Berti a contiguous view of the non-contiguous physical address space. This way, when accessing the last cache lines of a page, the first cache lines of the next page can be timely prefetched.

Next subsections describe both the training phase and the generation of prefetches in Berti. Then, we comment on the optimizations proposed on top of the Berti prefetcher.

A. *Training*

The goal of Berti is to calculate the best timely delta for the accesses in pages that are currently being accessed (hot pages). Hot pages are kept in a table named the *current pages* table (Figure 1), and that table is used to compute the best Berti delta for each of the currently accessed pages.

Berti only considers for the computation of the best delta accesses to cache lines that have not been recently demanded or prefetched, that is, that would probably cause a miss if the prefetcher would not have been active. To know if an access is a *potential miss*, Berti relies on a bit vector of accessed cache lines stored in each *current pages* table entry. The bit vector has a size of 64 bits (since in ChampSim simulator the cache line size is 64 bytes and the page size is 4KB) and each bit represents the offset of the cache line within the page. This way Berti knows if the cache line is accessed for the first time since the page became hot or not.

For every potential miss, Berti looks for the set of demand accesses that could have brought the line to cache on time if they were prefetching it when accessing the cache. Berti uses two structures for this: the *previous demand requests* table and the *previous prefetch requests* table (Figure 1). These structures store the page address (actually just a pointer to a hot page entry in the current page table), the cache line offset, and the issue time of the request. The *previous prefetch requests* table also stores a *completed* bit which indicates that the prefetch has been completed, and in this case the issue time field stores the time that the prefetch required to be resolved (its latency).

When a potential miss is resolved, its potential latency is calculated in the following way. In case of an actual miss, the latency is computed when the miss resolves, by looking up in the *previous demand requests* table the time when the request issued. In case of a hit in case due to a prefetched cache line, the latency is computed by looking at the latency of the completed prefetches in the *previous prefetch requests* table.

Once the latency is calculated, the *previous demand requests* table is searched in order to find the offsets that could have brought the cache line to cache in a timely manner, according to the obtained latency. The deltas with respect to these offsets are recorded in the *current pages* table, and a counter associated with each delta counts how many cache lines in the page found that delta. In the current implementation we store six deltas with their respective counters.

When a hot page is evicted from the *current pages* table the timely deltas are checked and the delta with higher count is selected as the Berti delta. The Berti delta is recorded in a new table called the *IP* table (Figure 1). This is a modification with respect to the original Berti prefetcher. In this work the Berti delta information is just stored per instruction pointer, and not also per page, as we observed similar best deltas across different pages accessed by the same set of instructions. Berti predicts best deltas for unseen pages based on the instruction pointer.

Since for some pages different memory instructions contribute to its accesses, Berti clusters instruction pointers that access the same page in order to predict the pattern accurately when any of those instructions access a new page. This is done in our particular Berti implementation by using a pointer in the IP table to the *current pages* table. Several entries in the IP table can point to the same entry in the *current pages* table, and all of them are updated with the new calculated best delta when the page is evicted from the *current pages* table, that is, the page becomes cold.

B. *Inferring prefetches*

The Berti prefetcher leverages the information in the *IP* table and the *current pages* table in order to asses if issuing prefetches or not. The *IP* table can store either the current Berti delta for that IP, or a pointer to the *current pages* table if the page is hot. In that case, the Berti delta is copied from the IP table to the *current pages* table.

The cache lines to be prefetched are calculated by adding the offset of the current request and the Berti delta. The prefetch is issued only if a demand access has not been previously issued for it (this information is available in the bit vector of accessed cache lines of the *current pages* table).

C. *Using an stride prefetcher for poorly accesses pages*

Our first optimization over the Berti prefetcher is the use of an stride prefetcher in case the pages are poorly accessed, namely, the page suffers just one or two accesses while they are hot. The stride is kept in the same field of the *IP* table as the Berti delta. A bit in the *IP* table indicates if the instruction accessed more than two cache lines in the last page or not.

D. *Linnea: Smooth page transitions*

Prefetchers that are trained with physical addresses are commonly not able to predict strides accross page boundaries, since contiguous virtual pages may be mapped *randomly* in physical space. The second and most important optimization over the Berti prefetcher addresses with problem by providing the illusion of being working with a contiguous address space. This way, page transitions do not imply missing prefetch opportunities for their first accesses.

We employ a *record pages* table aimed at correlating physical pages, i.e., it stores for any physical page accessed by an instruction (or cluster of instructions) the next accessed page. When a Berti delta applied on an access falls outside the current page boundaries, the next page is obtained from the *record pages* table and a prefetch on that new page is triggered. Depending on the direction of Berti, increasing or decreasing delta, the new page can be inferred as increasing or decreasing addresses, respectively.

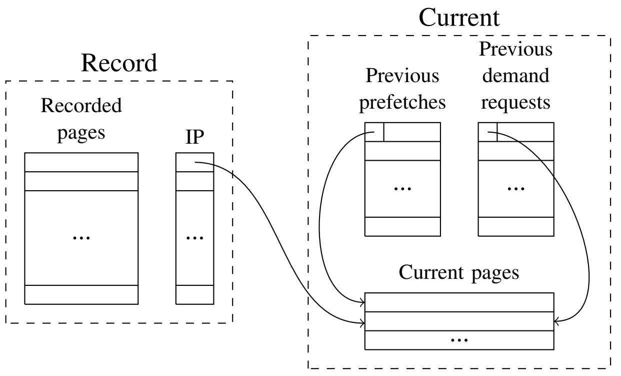

Fig. 1: Berti prefetcher overview

E. Overview of Berti

Figure 1 shows an overview of all the tables required for the Berti prefetcher. The tables in the *Current* dashed square collect information about pages being currently used. The tables in the *Recorded* dashed square collect information about cold pages. Fields stored in some tables that point to other tables are indicated with arrows. As we will discuss in Section V-A, the *previous prefetches* table has not been implemented due to the methodology limitations.

III. IP-based Entangling data prefetcher

The Entangling data prefetcher records correlations between to cache lines with a focus on timeliness. It keeps the history of accesses in a structure similar to the *previous demand requests* table. And on an access it entangles (i.e, correlates) that cache line with a previous cache line accessed by the same instruction a number of cycles before than the latency required to fetch the cache line.

The entangling data prefetcher uses a degree of two. The first cache line selected for prefetching is the one entangled to the current access, that is, that it is predicted to be accessed no earlier than the latency that it will take. The second cache line selected for prefetching is the next one accesses after the entangled one by the same instruction.

IV. BL∪E prefetcher

Our proposed prefetcher, BL∪E, combines i) a **B**erti prefetcher with the **L**innea optimization, ii) an IP-based **E**ntangling data prefetcher, and iii) a next-line prefetcher. We order the prefetchers from more accurate and less likely to provide any prefetch to less accurate and more likely to issue prefetchers.

The first prefetcher mechanism triggered is the IP-based Entangling data prefetcher, which will only issue a prefetcher if the cache line has been previously accessed by the same instruction. Then the Berti prefetcher is called. The Berti prefetcher is able to predict deltas for non-accessed pages, based on the previous accesses of the same instruction. Finally, a next-line prefetcher with a degree of two is triggered if the previous prefetchers did not manage to issue up to two prefetches. Although perhaps inaccurate, the next-line prefetcher is a good option when working at the last level cache, since wrong prefetchers do not entail a very high cost.

V. Methodology

We evaluate our prefetchers and compare them to state-of-the-art prefetchers using the version of the ChampSim simulator provided for the ML-Based Data Prefetching Competition. We normalize the IPC (instructions per cycle) obtained by each prefetcher to a baseline system without any prefetcher. All the prefetchers analyzed for the competition are placed in the last-level cache (LLC) without having any other prefetcher for the lower levels. Additionally, we provide results for the prefetcher that won the third data prefetching competition, IPCP [6], which settles at L1 and L2, and does not perform prefetches at LLC.

We evaluate our prefetchers using traces for the SPEC CPU 2017 applications. We collect statistics for 100 million instructions, after a 100-million-instruction warm-up period.

We have analyzed online versions of all the prefetcher considered in this work: a next-line prefetcher (NextLine) [1], both with degree one and two, a best-offset prefetcher (BOP) prefetcher [3], a correlation prefetcher of degree two (Corr2), named as *sisb* in the provided infrastructure, a BOP prefetcher combined with Corr2 (BOP+Corr2), provided as a state-of-the-art baseline in the ML-Based Data Prefetching Competition infrastructure, and four different flavors of our BL∪E prefetcher (Berti, Berti+Linnea, Berti+Linnea+Ent, and BL∪E). In addition, we analyze results for the offline versions of our prefetchers and some state-of-the-art prefetchers that could be easily ported to offline mode.

A. Methodology limitations

On one hand, the offline methodology provided in the competition does not allow to compute cache miss latencies. Miss latencies is a fundamental information to perform timely prefetches. Since the prefetchers for the competition work at the LLC miss latency variability is low. We note that the reported latencies were either 71 cycles (in most cases) or 171 cycles. As a consequence, we decided to hard-code the LLC miss latency as 171 cycles, such that we minimize late prefetches. Note that slightly early prefetchers are not a problem when considering LLC prefetching, the capacity of the LLC is quite large.

On the other hand, the offline methodology performs the training for a trace of accesses generated without issuing prefetches, and then predict blindly for the remaining of accesses. First, the training without considering the effect of your own prefetches may be inaccurate since prefetches alter the timing of subsequent memory accesses. Second, training stops after warm-up period in the off-line methodology, while in an online methodology training never stops. For these reasons the differences in performance of both methodologies are sometimes around 2%, on average. The more advanced is the prefetcher, the larger is the relative performance degradation of offline prefetching.

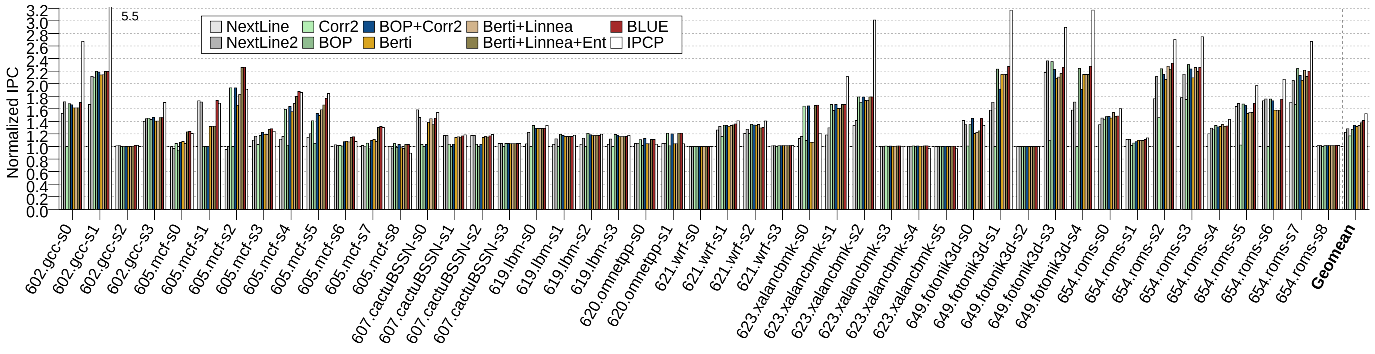

Fig. 2: Normalized IPC for SPEC CPU 2017 applications

TABLE II: Evaluation results (geometric mean)

Prefetcher	Cache Level	Online	Offline	Diff. (%)
NextLine	LLC	1.225835	1.218154	0.63
NextLine2	LLC	1.280341	1.274004	0.50
BOP	LLC	1.274494		
Corr2	LLC	1.167725	1.158600	0.79
BOP+Corr2	LLC	1.338013		
Berti	LLC	1.317987	1.311176	0.52
Berti+Linnea	LLC	1.334678	1.325316	0.70
Berti+Linnea+Ent	LLC	1.376534	1.358659	1.32
BL∪E	LLC	1.413864	1.391404	1.61
IPCP (64KB)	L1 & L2	1.520055		

VI. Evaluation results

Fig. 2 shows the normalized IPC for the prefetches evaluated via an online methodology and the set of SPEC CPU 2017 applications provided by the competition, and the geometric mean (also reported in TABLE II). A first observation derived from the results is that NextLine2 prefetcher can outperform BOP when prefetching for an LLC. The combination of BOP and Corr2 offers 34% improvements on IPC over a baseline without any prefetcher. Berti+Linnea improves BOP by a 6%. Together with Ent, a timely version of Corr2, improves BOP+Corr2 by 4%. Finally, BL∪E, which adds a NextLine2 prefetcher to Berti+Linnea+Ent improves the baseline provided in the competition by 8%.

Still we can see the importance of prefetching at lower cache levels with the IPCP prefetcher located at L1 and L2. It improves BL∪E performance by 11%.

TABLE II also shows offline results for most of the prefetchers evaluated. We can observe that there are some discrepancies in between the results given by both methodologies, reaching up to 2%, on average (see last column in TABLE II).

VII. Discussion

In this paper we present a prefetcher that it is not based on machine learning. However, the presented concepts, such as deltas or timeliness, are fundamental in order to have performant machine-learning models. With the current methodology provided in the competition, however, it is not possible to know if a prefetch will be late or timely. We believe that without that information prefetchers will lead to sub-optimal solutions.

On the other hand, the methodology does not considers lower level prefetchers. In the 3rd Data Prefetching Championship, where contestants could allocate their prefetcher budget at any cache level, it was demonstrated that was more important to allocate storage in lower cache levels [4], [6]. Not considering low-level prefetchers can also lead to sub-optimal solutions. The goal of this work was not to evaluate BL∪E in the presence of low-level prefetchers. We leave this interesting research line as future work.

Finally, it would be interesting to investigate the performance differences between the online and the offline methodologies.

VIII. Acknowledgments

This work was supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134).

References

[1] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 1st ed., 2009.

[2] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pp. 63–74, Feb. 2007.

[3] P. Michaud, "Best-offset hardware prefetching," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pp. 469–480, Mar. 2016.

[4] A. Ros, "Berti: A per-page best-request-time delta prefetcher," in *The 3rd Data Prefetching Championship*, June 2019.

[5] A. Ros and A. Jimborean, "The entangling instruction prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[6] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based hardware prefetching," in *The 3rd Data Prefetching Championship*, June 2019.