Berti: A Per-Page Best-Request-Time Delta Prefetcher

Alberto Ros
University of Murcia
aros@ditec.um.es

ABSTRACT

Prefetching data blocks into the caches comprising the memory hierarchy is a fundamental technique for designing high-performance computers. In fact, current systems implement prefetchers at every cache level. Timeliness is an essential property for getting the maximum performance from the prefetcher, as bringing the data early to cache can increase its miss ratio and requesting the data too late can lead to sub-optimal performance.

This paper presents Berti, a prefetcher that finds the delta that provides the best timeliness for memory blocks in each page. The prefetcher works in two modes: (i) on the first access to a block, in a certain period of time, the prefetcher issues a request for the next block according to the best delta found; (ii) for cold pages, a burst mechanism fetches blocks that cannot be reached adding the delta to the current accessed block.

1. INTRODUCTION AND MOTIVATION

A widely employed prefetching technique is to find a stride or delta that repeats for each memory instruction and to use that stride to prefetch next addresses [1]. These prefetchers are known as stride-based prefetchers. However, the stride can be difficult to find, as the order of accesses can be altered by the out-of-order core, the lower cache levels, or even the prefetchers of lower cache levels. Additionally, just prefetching the next address according to the stride may lead to late prefetches, resulting in sub-optimal performance.

These observations were leveraged by Michaud to propose the best-offset prefetcher (BOP) [2], which was the best performing prefetching technique in the 2nd Data Prefetching Championship. BOP finds the best delta for the accesses performed by an application, and applies it to the next accesses.

Our prefetching mechanism, highly inspired in BOP, is based on the observation that timeliness, and therefore the best delta, varies from page to page, and a global delta results in missing opportunities. As a consequence, our prefetching mechanism finds a unique delta per page that provides the best timeliness for its blocks and applies it to the next accesses. For those blocks that cannot be reached by the selected delta, a burst of prefetches is initiated on the first access to a cold page.[1]

Additionally, we use the instruction pointers (IP) in order to link them to the pages that they access, as the best delta tends to be stable for each instruction. In this way, when a new page is accessed and its delta is not known, the prefetcher can prefetch according to the instruction pointer. Instructions accessing the same pages are clustered and point to the same delta.

[1] A memory page that has not been accessed for a long time.

Finally, the confidence of the prediction is built depending on the matching with the information available. We keep the first offset (the block position in the page) accessed within each page. A match with the page and the first offset provides more confidence than a match only with the page. This confidence mechanism is inspired by the Bingo prefetcher [3].

In particular, the proposed prefetching mechanism builds on the following concepts: (i) per-page best delta for timely prefetches, (ii) burst of prefetches for cold pages starts, (iii) IP tracking and IP clustering to predict pages that have not been accessed yet, and (iv) confidence built as the current access matches better with the collected information.

2. THE BERTI PREFETCHER

The main goal of our prefetch technique is to collect two pieces of information about previously accessed pages, both of them independent of the order in which the accesses took place since, as mentioned in the introduction, they can arrive at any order. These two pieces of information are the blocks (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 memory blocks are prefetched. The Berti prefetcher has two prefetching modes: Burst and Berti. The Burst mode starts on the first access to a cold page, and it aims to prefetch the memory blocks accessed in between the first access to the page and the Berti delta. These blocks cannot be prefetched by the Berti mode, as they would require to be triggered by accesses in another page. The burst prefetches are expected to be late prefetches but, since we are bound to accesses within the same memory page, it is not possible to make them timely. The Berti mode is initiated when the burst finishes, and it issues prefetches according to the Berti delta, which are expected to be timely.

As a clarifying example, let us consider a memory page found in *429.mcf-217B* (SPEC CPU 2006). The information collected by the Berti prefetcher is shown in Table 1. The accessed blocks are represented in a bit vector, where 1 indicates that the memory block 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 block corresponds to six offsets before the current access.

In bold are represented both the first access to the page (the right one) and the block that would be prefetched on the first access according with the Berti delta (the left one). The three ones (1) that lay in between the first accessed block and the

Table 1: Example found in *429.mcf-217B*

Blocks accessed [0..63]	Berti
...11011**1**0110**1**1**1**0	-6

first Berti prefetched block (in red) are late prefetches that need to be covered by the Burst mode. The remaining of the accesses will be covered by the Berti mode and are expected to be timely.

Next subsections offer, first, details about how the information is collected and, then, how the prefetcher decides which blocks to prefetch based on that information.

2.1 Gathering information

The Berti prefetcher gathers two classes of data at run-time: information directed to predict which blocks to prefetch, such as the vector of accessed blocks and the Berti delta for each page, and meta-information which is used to locate the first information and to give a high or low confidence on the prediction, such as the page address, the instructions pointers, and the offset of the first accessed block.

2.1.1 Bit vector of accessed blocks

The Berti prefetcher collects information for the pages that are currently being accessed (hot pages). The information about the memory blocks accessed within a page is quite straightforward to collect. We use a bit vector of 64 bits (as in ChampSim simulator the block size is 64 bytes and the page size is 4KB), where each bit represents the offset of the block within the page. We keep the hot pages within a table, namely the *current pages* table, which matches the accessed blocks vector with the page address (Figure 1). On every access we add the offset of the block to the accessed blocks vector. In case an access for a cold page takes place, we add it to the *current pages* table and the information of the evicted page is recorded in a new table, the *recorded pages* table (Figure 1).

2.1.2 Berti delta

The goal of our prefetching mechanism is to calculate the best timely delta for the accesses in hot pages. We consider every access that would have caused a miss if the prefetcher would not have been active, namely *potential misses*. We leverage the accessed blocks vector in order to find if the block is accessed for the first time since the page became hot. If there is a hit in the cache, but we have not recorded a previous access to this block, then we assume that the block has been brought to the cache by one of our prefetches.

For every potential miss, we need to find the set of requests that could have brought the block to cache on time if they were prefetching it at their issue time. We require 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 block 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 obtained 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 block, 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 again in order to find the offsets that could have brought the block 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 blocks in the page found that delta. In the current implementation we store up to ten 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, along with the accessed blocks vector, is recorded in the *recorded pages* table.

2.1.3 Page address, offset of first access, and IP

Both the *current pages* table and the *recorded pages* table, which store the relevant information to perform the prefetches, are looked up using the page address. However, the *recorded pages* table can also be looked up using the information regarding the first access to the page. This gives more confidence in the prediction, as we will describe it in the next subsection, and allows the same page to have several entries in the *recorded pages* table. This information is initially recorded in the *current pages* table and moved to the *recorded pages* table when the page is evicted from the former table. Therefore, both tables need to be extended with an *offset of first access* field.

Additionally, IPs can be used in order to predict prefetches for new pages that have not been accessed yet, as the same instruction may frequently follow the same trend in its accesses. We also noted that for some pages different memory instructions contribute to its accesses. Therefore, we cluster instruction pointers that access the same page in order to predict the pattern accurately when any of those instructions access a new page.

In order to collect the clustered IP information, the *current pages* table is extended with a field storing the first IP that accesses a hot page. We also introduce an *IP* table (Figure 1) indexed by a hash function of the IP (least significant bits in the current implementation). The *IP* table only stores a pointer to an entry in the *recorded pages* table.

On the first access to a hot page the *first IP* field registers the IP and a pointer for this IP is created in the *IP* table. When a new IP accesses the same hot page, the *current pages* table is not modified, but the pointer of the first IP is assigned to the new IP entry. This way, clustered IPs point to the same entry in the *recorded pages* table.

2.1.4 Berti prefetcher scheme

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.

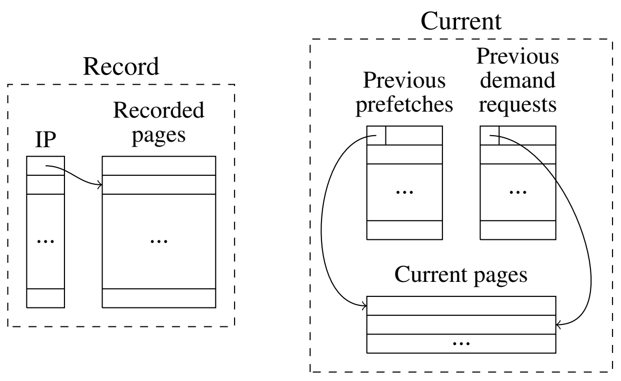

Figure 1: Berti prefetcher overview

2.2 Issuing prefetches

The Berti prefetcher leverages the information in the *IP* table, the *recorded pages* table, and the *current pages* table in order to asses if issuing prefetches or not. Depending on the matching of the current request with the information recorded, a different confidence level will be assigned, and a different prefetch mode, Burst or Berti, can be applied.

2.2.1 Confidence

The Berti prefetcher builds its confidence depending on the matching information with the current IP of the access, the current page address and the first offset that accessed the page. A match with the page address and the first accessed offset in the *recorded pages* table gives the higher confidence followed by a match with the IP of the current request and the first accessed offset. In both cases, when a match with the first accessd offset happens, the Burst mode is initiated.

In case of not finding a match with the first accessed offset and the IP or the page address, the prefetcher checks the *current pages* table, looking for the Berti delta. If its counter gives us some confidence (a value of 2 blocks finding this delta is employed in the submitted version), then we use the Berti delta of the current page. No burst is necessary in this case since the page is hot.

Otherwise, in case of no matching in the previous tables, we look for a match in the page address in the *recorded pages* table (the most recently allocated entry, since there may be several entries for the same page) or, in case of a mismatch, we try to find the entry in the *recorded pages* table pointed by the IP table. In both cases, we employ the Berti delta stored in the *recorded pages* table, and no Burst mode is initiated, as the confidence is and Burst is an aggressive technique.

Finally, if none of the previous matches happen, the prefetcher will opt for not issuing any prefetch.

2.2.2 Burst prefetches

The Burst mode is responsible for prefetching the blocks that cannot be prefetched with the Berti mode. If the Berti delta is too large many blocks may fall into this category. This mode starts on the first access to a cold page and if the offset of this access matches the first accessed offset recorded. Only the blocks that where accessed previously by the page are prefetched. This information is obtained from the bit vector of accessed blocks. And only blocks in between the current offset and the current offset plus the Berti delta are accessed. These prefetches are expected to be late prefetches, but they help to improve performance.

Burst prefetches do not need to be triggered all at once. We keep track of the last burst prefetch offset in the *current pages* table, allowing to split the burst across several accesses. In the current implementation we issue three Burst prefetches per access.

2.2.3 Berti prefetch

The Berti prefetches are expected to be timely prefetches. The block to be prefetched is 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 blocks of the *current pages* table). In case the Berti delta is obtained from the *recorded pages* table and there is a match in the first accessed block, the prefetch is only issued if the current block was accessed before (this information is available in the bit vector of accessed blocks of the *recorded pages* table). In case there is not a match with the first accessed block, the Berti prefetch is issued.

3. MEMORY REQUIREMENTS

The Berti prefetcher has been employed for all cache levels. They configurations only differ in the size of the *recorded pages* table, which is larger for lower level prefetchers. The reason for this decision is that a good timely prefetching technique would basically require a good L1 prefetcher such that blocks are in the L1 when the processor requests them. Table 2 shows the memory requirements of the Berti prefetcher for each cache level and the total memory requirements (less than 64KB of memory per core).

The *current pages* table is a fully associative structure which is looked up with the page address (52 bits). It also contain the following fields: the IP index (10 bits), the bit vector of accessed blocks (64 bits), the offset of the first access (6 bits), ten timely deltas and their counters (7 + 6 bits each), the current burst offset (6 bits), and the least recently used (LRU) information of the table.

The *previous demand requests* table is a circular queue. Each of its entries stores a pointer to a *current pages* table entry (depends in the *current pages* table entries, 6 bits in our case), the offset of the request (6 bits), and the issue time of the request (16 bits are used as an approximation).

The *previous prefetches* table is also a circular queue. It stores a pointer to a *current pages* table entry (6 bits), the offset of the request (6 bits), its issue time or latency (16 bits), and a completed bit.

The *recorded pages* table is a fully associative table where each entry stores part of the page address (32 bits), the bit vector of accessed blocks (64 bits), the offset of the first access (6 bits), the Berti delta (7 bits), and the LRU information.

Finally, the *IP* table is a direct mapped table that stores a pointer to an entry in the *recorded pages* table.

4. EVALUATION RESULTS

We evaluate both configurations with one core and four cores with SPEC CPU 2017 applications. For one core, we evaluate the applications that have, without any prefetching,

Table 2: Memory requirements per core

Cache	Structure	Number of entries	Bytes per entry	Total (bytes)
L1D	Current pages	2^6	$(52+10+64+6+(7+6)*10+6+6)/8$	2192
L1D	Previous demand requests	2^{10}	$(6+6+16)/8$	3584
L1D	Previous prefetches	2^9	$(6+6+16+1)/8$	1856
L1D	Recorded pages	$2^{10}+2^8+2^7$	$(32+64+6+7+11)/8$	21120
L1D	IP	2^{10}	$11/8$	1408
L2C	Current pages	2^6	$(52+10+64+6+(7+6)*10+6+6)/8$	2192
L2C	Previous demand requests	2^{10}	$(6+6+16)/8$	3584
L2C	Previous prefetches	2^9	$6/8+6/8+2+1/8$	1856
L2C	Recorded pages	2^9+2^8	$(32+64+6+7+9)/8$	11424
L2C	IP	2^{10}	$10/8$	1280
LLC	Current pages	2^6	$(52+10+64+6+(7+6)*10+6+6)/8$	2192
LLC	Previous demand requests	2^{10}	$(6+6+16)/8$	3584
LLC	Previous prefetches	2^9	$(6+6+18+1)/8$	1856
LLC	Recorded pages	2^8+2^7	$(32+64+6+7+8)/8$	5664
LLC	IP	2^{10}	$9/8$	1152
Total	Berti prefetcher			$(63.4KB)$ 64944

at least one miss per kilo instructions (MPKI) at the LLC. For four cores, several random mixes from all the SPEC CPU 2017 applications have been employed. Applications run for 200M instructions after a 50M instructions warm-up.

We have analyzed a next-line prefetcher (NextLine) [1], an stride prefetcher (Stride) [1], a Signature Path prefetcher (SPP) [4], a best-offset prefetcher BOP prefetcher [2], and a Kill the Program Counter prefetcher (KPCP) [5]. Simulations with the ported version of the BOP prefetcher did not offer the expected results (not better than SPP), and therefore, we opt for not showing them in the evaluation.

Table 3 shows the geometric mean for the configurations analyzed. We first employed a NextLine prefetcher of all cache levels. When we replaced the L2C prefetcher with a more elaborated prefetcher. The KPCP is the state of the art prefetcher with better results for the one core configurations, while for the four core configuration the best results are obtained by the Stride prefetcher. Berti outperforms KPCP for both configurations and obtain similar figures than SPP for the four core configuration. When applying Berti to all levels, however modest improvements are obtained. Surprisingly, Berti does not performs as expected for the LLC, and simple prefetchers are preferable for that cache level, being able to give its memory budget to the L1D prefetcher (Berti+). The reason is that the average miss latency at the LLC is larger than at lower levels and many pages will use very large Berti deltas, missing a lot of opportunities for late prefetches.

Table 3: Evaluation results (geometric mean)

Configuration (L1D, L2C, LLC)	1 core	4 cores
NextLine, NextLine, NextLine	1.268853	1.104525
NextLine, Stride, NextLine	1.294615	1.114741
NextLine, SPP, NextLine	1.308339	1.104330
NextLine, KPCP, NextLine	1.314242	1.103204
NextLine, Berti, NextLine	1.321133	1.104235
Berti, Berti, Berti	1.334763	1.108715
Berti+, Berti, NextLine	1.347102	1.118640
Berti+, Berti, None	1.330342	1.126840

5. DISCUSSION AND FUTURE WORK

In the submitted version of the prefetcher we employ fully associative structures, as there are no associativity limits in the championship rules. However, an actual implementation of this prefetcher will use set-associative structures.

Our implementation does not explores late prefetches. However, it has been seen that this may be a limitation for last level caches. Future work can account for late prefetches in order to calculate the Berti delta. to prefetch

Finally, the Berti prefetcher does not uses confidence counters as other prefetchers found in the literature [5]. Confidence counters are useful in order to learn about the goodness of the performed prefetches and act as consequence. Future versions of the Berti prefetcher will benefit from confidence counters, as we found a large amount of useless prefetches in our simulation results.

6. ACKNOWLEDGMENTS

"This work was supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant RTI2018-098156-B-C53"

7. REFERENCES

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

[2] P. Michaud, “Best-offset hardware prefetching,” in *22th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2016.

[3] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo spatial data prefetcher,” in *25th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019.

[4] J. Kim, P. V. Gratz, and A. L. N. Reddy, “Lookahead prefetching with signature path,” in *2nd Data Prefetching Championship*, June 2015.

[5] J. Kim, E. Teran, P. V. Gratz, D. A. Jimenez, S. H. Pugsley, and C. Wilkerson, “Kill the program counter: Reconstructing program behavior in the processor cache hierarchy,” in *Int’l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 2017.