Berti: an Accurate Local-Delta Data Prefetcher

Agustín Navarro-Torres*, Biswabandan Panda†, Jesús Alastruey-Benedé*,
Pablo Ibáñez*, Víctor Viñals-Yúfera*, and Alberto Ros‡

*Dept. Informática e Ingeniería de Sistemas - I3A, Universidad de Zaragoza, Zaragoza, Spain
†Dept. of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India
‡Computer Engineering Department, University of Murcia, Murcia, Spain
Email: agusnt@unizar.es, biswa@cse.iitb.ac.in, jalastru@unizar.es, imarin@unizar.es, victor@unizar.es, aros@ditec.um.es

***Abstract*—Data prefetching is a technique that plays a crucial role in modern high-performance processors by hiding long latency memory accesses. Several state-of-the-art hardware prefetchers exploit the concept of deltas, defined as the difference between the cache line addresses of two demand accesses. Existing delta prefetchers, such as best offset prefetching (BOP) and multi-lookahead prefetching (MLOP), train and predict future accesses based on *global* deltas. We observed that the use of global deltas results in missed opportunities to anticipate memory accesses.**

In this paper, we propose Berti, a first-level data cache prefetcher that selects the best *local* deltas, i.e., those that consider only demand accesses issued by the same instruction. Thanks to a high-confidence mechanism that precisely detects the timely local deltas with high coverage, Berti generates accurate prefetch requests. Then, it orchestrates the prefetch requests to the memory hierarchy, using the selected deltas.

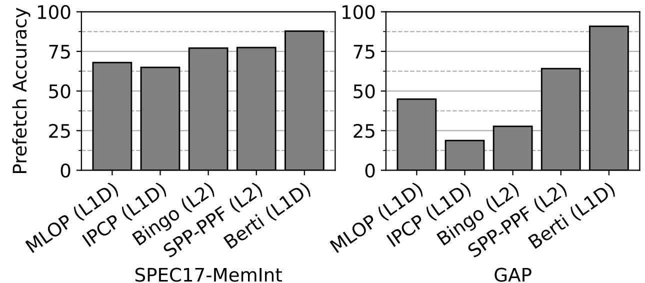
Our empirical results using ChampSim and SPEC CPU2017 and GAP workloads show that, with a storage overhead of just 2.55 KB, Berti improves performance by 8.5% compared to a baseline IP-stride and 3.5% compared to IPCP, a state-of-the-art prefetcher. Our evaluation also shows that Berti reduces dynamic energy at the memory hierarchy by 33.6% compared to IPCP, thanks to its high prefetch accuracy.

***Keywords*-data prefetching; hardware prefetching; first-level cache; local deltas; accuracy; timeliness**

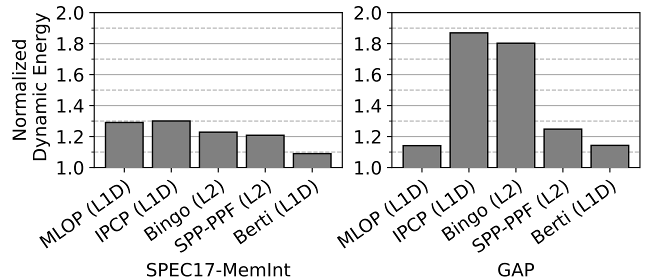
I. Introduction

Data prefetching techniques play an important role in hiding long-latency memory accesses. Hardware prefetchers learn memory access patterns and fetch data into the cache hierarchy before time so that future memory accesses get cache hits. Data prefetching techniques can be employed either at the private first-level data cache (L1D), second-level cache (L2), or at the shared last-level cache (LLC).

Most of the recently proposed storage-efficient spatial prefetchers target L2 [13], [17], [35], [38], [50]. Exceptions are the multi-lookahead offset prefetching (MLOP) [48] and the instruction pointer classifier-based prefetching (IPCP) [40], which are L1D prefetchers. It is well known that an L1D prefetcher provides better performance than an L2 prefetcher as the prefetched lines are brought into L1D and not till L2. In addition, an L1D prefetcher sees unfiltered memory access patterns and can predict the future



(a) Accuracy of L1 and L2 prefetchers.



(b) Dynamic energy consumption normalized to no prefetching.

Figure 1. Prefetch accuracy and dynamic energy consumption of the memory hierarchy for state-of-the-art prefetchers (IPCP [40], MLOP [48], SPP-PPF [17], and Bingo [13]) averaged across single-threaded traces from memory-intensive SPEC CPU2017 [55] and GAP [14] workloads.

accesses better than an L2 or LLC prefetcher. L1D also sees a sequence of virtual addresses as compared to physical addresses at the L2 and LLC, which can facilitate cross-page prefetching [24]. Also, compared to L1D, additional contextual information is not easy to propagate to L2 and LLC, such as instruction pointer (IP) [36], which is usually available at the L1D (e.g., Intel's IP-stride at the L1D [20]). However, designing a high-performance L1D prefetcher is hindered by (i) storage overhead, (ii) starved L1D bandwidth, (iii) L1D pollution because of inaccurate prefetching, and (iv) narrow scope for aggressive prefetching because of the limited size of the prefetch queue (PQ) and the miss status holding registers (MSHR).

State-of-the-art data prefetchers push the limit of single-thread performance with average performance boosts of

3% to 5% [13], [17], [35], [40]. However, as shown in Figure 1(a), these prefetchers load a large number of useless blocks, ranging from 22.6% to 35.1% for SPEC CPU2017 and from 35.8% to 81.2% for GAP workloads, which results in sub-optimal performance and additional dynamic energy consumption [34]. Figure 1(b) shows that state-of-the-art prefetchers significantly increase the dynamic energy consumption at the memory hierarchy (caches and DRAM) up to 30.1% and 86.9% for SPEC CPU2017 and GAP workloads, respectively.

Our proposal, Berti, provides an accuracy of almost 90%, which translates into a dynamic energy overhead of only 9.0% and 14.3% for SPEC CPU2017 and GAP, respectively.

Our approach. We ask the following simple question in designing our approach: "*for an L1D access to address X, what is the* timely and accurate *delta (d) that should be used for prefetching?*" The best offset prefetcher (BOP) inspires us to ask this question [38]. However, our approach is different from BOP and other offset prefetchers [29], [48]. Our key observation is that the best delta for access is dependent on the local contextual information, such as an instruction pointer (IP), and it varies based on the context (e.g., the best delta for IP_X is different from IP_Y). We argue that prefetching based on global (context-agnostic) deltas results in missing opportunities [39].

We propose Berti, a cost-effective, per-IP *best request time* delta L1D prefetcher that makes a strong case for timeliness and accuracy. For each IP, Berti learns the deltas that result in timely prefetch requests, and issue prefetch requests only for the deltas predicted to provide high coverage, which translates to overall high prefetch accuracy. Sited at the L1D and seeing all virtual addresses generated by the processor, Berti orchestrates the prefetch requests to the memory hierarchy. Our Berti prefetcher is inspired by Berti from DPC-3 [46].

Accurate and timely local deltas. We define local delta as the difference in cache line addresses between two demand accesses that are issued by the *same IP*. The definition of delta differs from the definition of stride, being the later the difference between addresses of *consecutive* load accesses with the same IP. For example, an IP_X that accesses the following cache line addresses: X, X+2, X+4, X+6, sees a sequence of strides as follows: +2, +2, and +2. In this case, the stride is 2. However, access X+6 sees the following deltas: +6, +4, and +2. Figure 2 shows an example differentiating strides, local deltas, and timely local deltas. If the goal of a prefetcher is to *cover* address 15, then the prefetcher can initiate prefetching with deltas +3, +5, and +8 whenever it sees the demand accesses to addresses 12, 10, and 7, respectively. However, if we consider time to prefetch address 15, then deltas of +3, +5, and +8 will not completely mitigate the L1D miss latency, as they will be late prefetch requests. Instead, if a prefetcher issues a request for address 15 with deltas of +10 or +13 on demand accesses to address 5 or 2, respectively, it can prefetch address 15 well ahead of time.

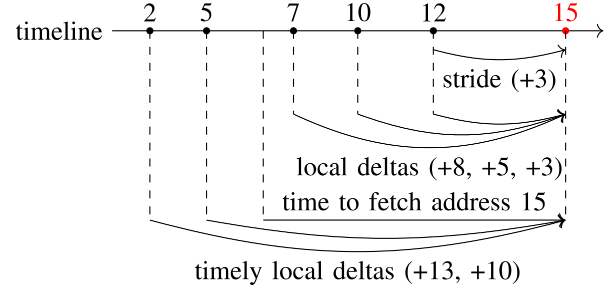


Figure 2. Strides, local deltas, and timely local deltas. The values on the timeline (2, 5, 7 ...) represent the addresses referenced by the same instruction.

With Berti, we find the *timely* local deltas, and compute its respective coverage. We prefetch using deltas that used to show high coverage, which translates to overall high prefetch accuracy as we show in this work. We call these deltas the accurate and timely deltas.

Contributions. We make the following key contributions:

- We motivate the need for a local L1D delta prefetcher to achieve a high coverage with accurate and timely prefetch requests (Section II).
- We propose Berti, a highly-accurate prefetcher that learns timely local deltas and predicts their coverage with high confidence. High accuracy (almost 90%) is achieved by only triggering prefetch requests for deltas with high coverage. Berti incurs a storage overhead of only 2.55 KB (Section III).
- Our evaluations show that for SPEC CPU2017 and GAP workloads, Berti consistently outperforms a baseline IP-stride prefetcher (8.5% average speedup). It also surpasses the highly-competitive state-of-the-art IPCP L1 prefetcher (3.5% average speedup), which is a significant uplift in performance, pushing further the limits of hardware prefetchers. Berti is equally effective with multicore heterogeneous mixes. Overall, Berti provides the best trade-off in performance, storage overhead, and energy consumption of the memory hierarchy (Section IV).

II. Recent Works and Motivation

A. Recent advances in data prefetching

Data prefetching plays an important role in designing high performance processors. Recent developments in this field mainly come from the last two data prefetching championships, DPC-2 [2] and DPC-3 [7], co-located with ISCA 2015 and ISCA 2019, respectively.

Best offset prefetching (BOP). The winner of DPC-2 is a degree-one L2 prefetcher that finds an offset that provides the maximum likelihood of future use at the L2 cache [38]. An offset of k means that a cache line is k cache lines away from the current demand address. BOP takes timeliness into account while selecting the best offset per application

phase. Multi-lookahead offset prefetching (MLOP) [48] is an extension on BOP that is motivated by Jain's Ph.D. thesis [29]. MLOP considers multiple lookaheads for each offset and selects the offset and lookahead covering a specific cache miss. Both BOP and MLOP treat the demand addresses in isolation, and for each demand access, trigger prefetch requests based on the prefetch offset[1]. In general, MLOP provides better prefetch coverage than BOP.

Variable Length Delta Prefetching (VLDP). This spatial data prefetcher uses multiple histories of deltas between successive cache lines observed within an operating system (OS) page to predict the future memory accesses in other OS pages [50]. One of the key features of VLDP is that it uses multiple prediction tables and makes predictions based on different lengths of history in terms of deltas.

Signature path prefetching (SPP). This state-of-the-art delta prefetcher predicts irregular strides at the L2 cache [35]. SPP works by relying on the signatures (hashes of consecutive strides) observed within an OS page to index into a table that predicts future deltas. SPP uses a lookahead mechanism that recursively finds out deltas to prefetch until a delta falls below a *confidence*. Perceptron prefetch filtering (PPF) is a filter that further improves the effectiveness of SPP by deciding whether to prefetch into L2 or not [17]. In general, SPP combined with PPF (SPP-PPF) provides better prefetch coverage than VLDP.

Bingo. This L2 prefetcher makes a case for associating spatial access patterns to both short (such as IP) and long events (such as IP, IP+offset, and memory region) and selecting the best pattern for prefetching [13]. A key point of Bingo is the use of only one hardware table for both short and long events. This table enables multiple predictions from a single entry, providing better coverage than single-event prefetching. In general, Bingo outperforms VLDP and SPP-PPF for SPEC CPU2017 traces. However, it requires significantly more storage than VLDP and SPP-PPF.

Instruction pointer classifier prefetching (IPCP). The winner of DPC-3 is a state-of-the-art L1D data prefetcher that is composite in nature [40]. It classifies an IP into three classes: constant stride (CS), complex stride (CPLX), and global stream (GS). IPCP uses three lightweight prefetchers that issue prefetch requests according to the IP class. If it fails to classify an IP into one of the three classes, it uses a next-line prefetcher.

B. *Motivation: why a new delta prefetcher?*

Why not a global delta prefetcher? We observe that finding the best delta for an entire application results in missing opportunities because the best delta varies based on the program context, e.g., an IP or the OS page. Figure 3 shows the best deltas selected by BOP (red line) and Berti

[1]For BOP and MLOP, we use the term *global delta* instead of *offset* for the rest of the paper.

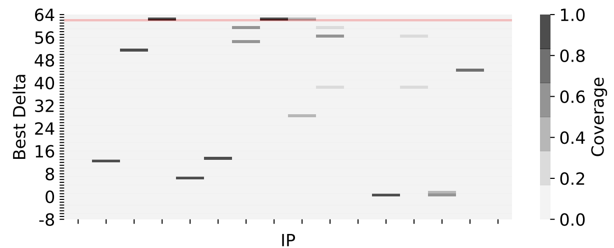


Figure 3. Best delta selected by BOP (based on global deltas) and Berti (based on per-IP local deltas) for `mcf-1554B`. Prefetch coverage is shown in grayscale. BOP selects +62 as the best delta (red line), which is not always accurate and provides a coverage of only 2%.

(gray lines) for different IPs of the `mcf-1554B` benchmark. We can see that the best delta is different for distinct IPs making a strong case for prefetching local, timely deltas instead of a global best delta (oblivious to per-IP best deltas). We can also see that the global delta (+62) as selected by BOP does not cover all cache accesses, and it is not the best delta. For `mcf-1554B`, BOP provides coverage of only 2%, whereas Berti, that selects local deltas (per IP), provides better coverage as shown in Figure 3. As we will see in Section IV, the use of a global delta may be beneficial in some cases (e.g., in `CactuBSSN`), but this is not the common case.

Why not existing local L1D prefetchers? A conventional IP-stride prefetcher covers consecutive constant strides and not necessarily timely deltas. For example, IP `0x401cb0` from `lbm-2676B` generates the following stride sequence: +1, +2, +1, +2, ... +1, +2. For this pattern, an IP-stride prefetcher will provide zero coverage and will not gather enough confidence to prefetch either with stride +1 or +2. IPCP's CPLX prefetcher will be able to detect this pattern. However, IPCP ignores the timeliness of prefetching. In contrast to IP-stride and IPCP's CPLX prefetcher, a more timely, accurate, and flexible approach would be to prefetch with deltas +3 or +6 that provide 100% coverage. Moreover, for the irregular stride sequence: -1, -5, -2, -1, -4, -1 associated to IP `0x402dc7` from `mcf-1554B`, IPCP's CPLX prefetcher fails to predict a pattern through its lookahead based on confidence. However, a local delta prefetcher with a delta of -1 can provide better coverage.

Effect of out-of-order loads at the L1D. In an out-of-order processor, memory accesses get reordered due to out-of-order scheduling. Hence, the training of a delta prefetcher may be affected by the ordering of memory accesses. Let's consider a loop with a single IP accessing memory addresses 1, 2, 3, 4, 5, 6, and 7 with constant strides of +1, +1, +1, +1, +1, +1. An out-of-order processor can reorder, for example, the accesses to addresses 2 and 3, resulting in the following sequence of addresses: 1, 3, 2, 4, 5, 6, 7 and strides +2, -1, +2, +1, +1, +1 at the L1D. This cannot be covered by an IP-stride or an IPCP's CPLX prefetcher unless a specific mechanism can provide the commit order [24], [56]. However, timely

deltas have the important property of seeing past accesses already in order, thus there is no requirement of such in-order commit mechanisms. Indeed, the last three accesses in our example will see the following past deltas: address 5 will see +4, +2, +3, +1, address 6 will see +5, +3, +4, +2, +1, and address 7 will see +6, +4, +5, +3, +2, +1 that is, all possibilities regardless of their order. The prefetcher then can choose the timely deltas that can provide the best coverage from these set of values.

III. Berti: A Local-Delta Prefetcher

Berti is a data prefetcher sited at the L1D, where it can see all the requests generated by the processor and orchestrate the prefetch requests to the memory hierarchy. Berti makes a strong case for prefetch accuracy. For each IP, it selects the deltas[2] that are timely and computes their respective local coverage. High accuracy is achieved by only using deltas with high coverage. Additionally, Berti is trained with virtual addresses, which helps in finding larger deltas and facilitates cross-page prefetching. Next, we describe how Berti performs training and prediction. Then, we propose a simple and cost-effective hardware implementation.

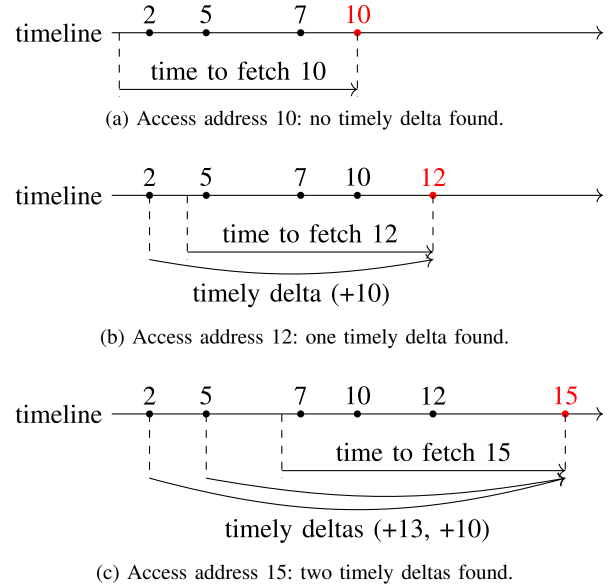
A. Training the prefetcher

The goal of the training mechanism is to estimate the coverage of each seen delta, considering only those deltas that would result in a timely prefetch. The training consists of the following actions: measuring fetch latency, learning timely deltas, and computing the coverage of the deltas.

Measuring fetch latency. In order to learn the deltas that are *timely* it is necessary to measure the time required to fetch data to the L1D, i.e., the L1D miss latency. This measurement is performed for any cache line in L1D, both for demand misses and prefetch requests. Computing latency for prefetch requests is fundamental because, in an ideal scenario, there would not be L1D misses but just L1D hits due to timely prefetch requests. In addition, the latency of prefetch requests may be larger than the latency of demand requests due to prefetch queue (PQ) contention or L1D port contention. Fetch latency can be measured by keeping a timestamp for any L1D miss inserted into the MSHR and any prefetch request inserted into the PQ. On an L1D fill, the latency is simply computed by subtracting the stored timestamp from the current one.

Learning timely and accurate deltas. Once the fetch latency is obtained for each L1D fill, our prefetcher can precisely learn timely deltas, given that the history of accesses and timestamps by the same IP is recorded. By searching in the history of recent accesses and comparing the timestamp of each previous access with the timestamp when a prefetch should be issued to be timely, the accesses that would trigger timely prefetch requests are detected. Deltas are then

[2]For the rest of the paper, unless specified we use the terms delta and local delta interchangeably.



(a) Access address 10: no timely delta found.

(b) Access address 12: one timely delta found.

(c) Access address 15: two timely deltas found.

Figure 4. Learning timely deltas.

computed by subtracting the address of each timely request in the history from the current address. Figure 4 depicts how timely deltas are detected. All addresses represented in the timeline are accessed by the same IP. When address 10 is demanded and its fetch latency computed (Figure 4a), the history of accesses for that IP is searched, from the point in time, a timely prefetch should have been triggered. In this case, no previous accesses are found. After accessing address 12 and computing its fetch latency (Figure 4b), a timely delta corresponding to address 2 is found. That is, address 2 should initiate the prefetch request for 12 in order to be timely. The timely delta +10 is therefore learned. Similarly, when computing the latency for access 15 (Figure 4c), two deltas, +10 and +13, are detected as timely.

Berti triggers the procedure to learn timely deltas for each miss that would have occurred in the baseline, which translates to two scenarios. First, when a demand miss fills the L1D with the requested data. Second, when a cache line brought into L1D by a prefetch request is demanded (i.e., misses that would have occurred without a prefetcher). Berti does not learn deltas on a cache fill caused by a prefetch request since its demand time is not known. Therefore, it is necessary to keep the latency of prefetch requests until the core demands the cache line.

Computing the coverage of deltas. On every search in the history, Berti obtains a set of timely deltas. Deltas that frequently appear in the searches would cover a significant fraction of misses, while deltas that rarely appear would result in low coverage. It is easy to compute the coverage by dividing the number of occurrences of a delta by the number of searches in the history. For example, in Figure 4, after three accesses, the delta +10 has the higher coverage, being in two out of three searches (66.7%). If the same

access pattern continues, the delta +10 will reach close to 100% coverage. It is important to note that this local (per IP) coverage translates into accuracy. If a delta covers 100% of cache lines, since each access-delta pair results in only one prefetch request, that delta will bring 100% accuracy.

B. Prediction: issuing prefetch requests

Once we know the deltas and their associated coverage, we can orchestrate the prefetch requests across the cache hierarchy. Based on both the coverage of each delta and the L1D MSHR occupancy, we decide which deltas to use and till which cache level to prefetch. We use four watermarks to decide where to issue the prefetch requests. If the coverage of a delta is above a *high-coverage* watermark and the L1D MSHR occupancy is below the *occupancy* watermark, then prefetch requests using that delta get filled at all the cache levels till L1D. Otherwise, if the coverage is above a *medium-coverage* watermark, irrespective of the L1D MSHR occupancy, prefetch requests get filled till L2. Finally, if the coverage is above a *low-coverage* watermark, requests get filled only in the LLC.

To generate a prefetch request, we add the selected delta to the address of the current access and the resulting address is inserted in the PQ. Requests in the PQ are processed in a first-in-first-out (FIFO) order. Since our prefetcher is trained with virtual addresses, the generated prefetch requests are also in the virtual address space. A prefetch request obtains the physical address from the L2 translation look-ahead buffer (STLB). If the translation misses in the STLB, the prefetch request is dropped. If the translation is obtained, the prefetch request checks if the target block is already present in the cache it wants to fill. In case of a miss, the block is prefetched, and the request is inserted into the MSHR.

C. Hardware implementation

As outlined in Figure 5, Berti can be implemented with a small hardware budget and using simple structures and logic. Next, we describe the structures required to train the prefetcher and decide on the prefetch requests to issue to each cache level.

Measuring fetch latency. In order to be able to measure the fetch latency, the MSHR is extended with a 16-bit field (represented in Figure 5 in gray) that stores a timestamp on a demand L1D miss. Similarly, the PQ is also extended with an analogous field that stores the timestamp when a new prefetch request is added. The timestamp can be obtained from the clock of the local processor [47] or any other metric to approximate time (e.g., number of cache accesses). In our implementation, we use the former. When a prefetch request misses L1D, the timestamp is transferred from the PQ to the newly allocated MSHR entry. On an L1D fill, the latency of the request can be computed with a simple subtraction. The latency is stored using 12 bits. If an overflow is detected when computing the latency, it is set to zero, and therefore not

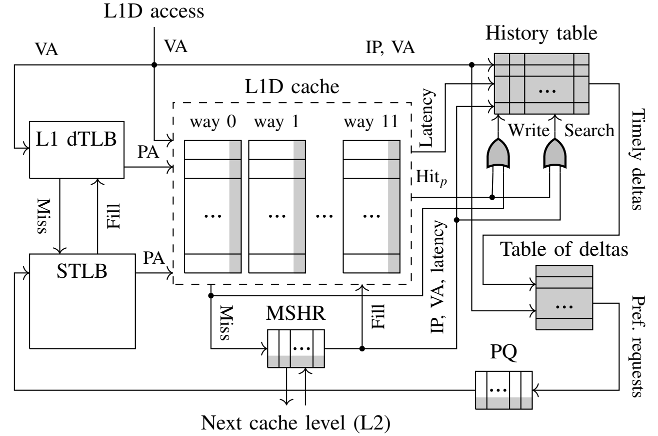


Figure 5. Berti design overview. Hardware extensions are shown in gray.

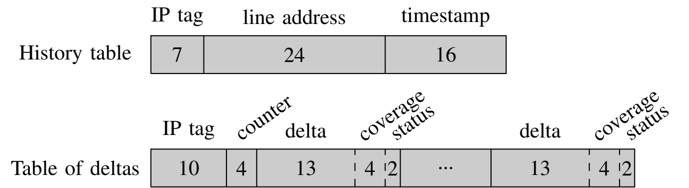


Figure 6. *History table* and *Table of deltas* entry format.

considered for learning timely deltas. Based on our empirical results, on average across GAP and SPEC CPU2017 traces, we see 1.08 overflows per kilo L1D fills.

Learning timely deltas. To be able to learn timely deltas, the most recent accesses need to be tracked. The *History table* (see Figure 5) records that information and is organized as an 8-set, 16-way cache with a FIFO replacement policy and indexed and searched with the IP. The format of each entry in the history table is depicted in Figure 6. Each entry keeps a tag corresponding to the seven least significant bits of the IP (after removing the bits used for indexing the cache), the 24 least significant bits of the target cache line address, and a 16-bit timestamp. A new entry is inserted in the history table (*Write* port in Figure 5) either on-demand misses (*Miss* arrow from the L1D in Figure 5) or on hits for prefetched cache lines (Hit_p in Figure 5). The virtual address (VA) and the IP (*IP, VA* arrow in Figure 5) are stored in the new entry along with the current timestamp (not shown in the figure).

The search for timely deltas (*Search* port in Figure 5) is performed either on a fill due to a demand access (*Fill* arrow from the MSHR in Figure 5) or on a hit due to a prefetched cache line (Hit_p in Figure 5). In the first case, the search is done using the information from the MSHR (*IP, VA, latency* arrow in Figure 5). In order to enable the search on L1D hits, we keep the latency of the prefetch request (12 bits) along with each entry in the L1D (see Figure 5 L1D shadow part). Alternatively, an L1D shadow tag could be employed. A latency field set to zero indicates either an overflow when

computing the latency or an already demanded cache line. In that case, a search in the history table is not performed. Otherwise, the search is done when the demand hit takes place, using the stored latency (*Latency* arrow in Figure 5), which is reset after the search. On every search, the 16-ways of the history table are looked up for a matching IP tag. A maximum of eight timely deltas, the ones corresponding to the youngest entries that would result in timely prefetch requests, are collected.

Computing the coverage of deltas. The results of each search in the history table (*Timely deltas* arrow in Figure 5) are accumulated in the *Table of deltas*, a 16-entry fully-associative cache with a FIFO replacement policy. The format of each entry in the table of deltas is depicted in Figure 6. Each entry consists of a 10-bit tag (based on hash function of the IP), a 4-bit counter, and an array of 16 deltas, each of them containing the delta itself (13 bits), the coverage (4 bits), and the status (2 bits) indicating till which cache level to prefetch. The counter is increased on each search in the history table. For each timely delta found during the search, its coverage counter is increased. When the counter overflows (its value increases to 16), we compute the coverage. Deltas that cross the *high-coverage* watermark (65% of coverage, i.e., a coverage value higher than 10) set their status to $L1D_pref$. Deltas in between the *high-coverage* watermark and the *medium-coverage* watermark (between 65% and 35%, i.e., a coverage value lower or equal than 10 and higher than 5) set the status to $L2_pref$. The maximum number of deltas selected for any of those status is bounded to 12. The remaining deltas' status is set to No_pref (i.e., do not issue prefetch requests for this delta). Once the status is set, the counter and the array of confidences are reset, and a new learning phase begins.

While warming-up the status fields, prefetch requests are also issued if at least eight deltas have been gathered, increasing the *high-coverage* watermark to 80%, as with just four deltas the prefetcher needs more confidence. Our empirical study shows that using watermarks higher that 65% leads to high accuracy.

Although Berti opens the possibility of prefetching to LLC only for low-coverage deltas, our evaluation showed no performance improvements when choosing this option. Hence, we set the *low-coverage* watermark to 35% (equal to the *medium-coverage* one), to disable prefetching to LLC only.

In order to constantly learn new deltas, evictions of deltas may be necessary. On the arrival of a non stored delta, deltas with less than 50% coverage in the previous phase are candidates for evictions in the current phase. To this end, if the coverage when selecting the $L2_pref$ status is lower than 50%, the status is set to $L2_pref_repl$. The eviction policy selects the delta with lower coverage whose status is $L2_pref_repl$ or No_pref. In case no such delta exists, the new delta is discarded.

Table I
STORAGE OVERHEAD OF BERTI.

Structure		**Storage**
History table	8-set, 16-way (128-entry) cache, FIFO replacement policy. Each set: 4 bits (replacement policy). Each entry: 7-bit IP tag, 24-bit address, 16-bit timestamp	0.74 KB
Table of deltas	16-entry, fully-associative, 4-bit FIFO replacement policy. Each entry: 10-bit IP tag, 4-bit counter, and an array of 16 deltas (13-bit delta, 4-bit coverage, 2-bit status)	0.62 KB
PQ + MSHR	16+16 entries, 16-bit timestamp per entry	0.06 KB
L1D	768 cache lines, 12-bit latency per line	1.13 KB
Total		**2.55 KB**

Issuing prefetch requests. On every L1D access, the table of deltas is searched looking with a matching IP (*IP, VA* arrow pointing to the table of deltas in Figure 5). Since we use an L1D with two read ports and one write port, the table of deltas requires three search ports. The deltas with status $L1D_pref$ or $L2_pref$ are added to the current VA to form the prefetch requests that are inserted into the PQ (*Pref. requests* arrow in Figure 5). Those prefetch requests get filled into all cache levels till L1D when the status is $L1D_pref$ and the MSHR occupancy is below 70% (the *occupancy* watermark). Otherwise, prefetch requests get filled till L2.

Storage overhead. Berti does not require any complex operation (e.g., multiplications) nor complex logic. Our history table has two read ports and one write port. The latency of this structure is two cycles, based on CACTI-P [37]. Since prefetching training is out of the critical path of memory accesses, the history table does not affect the cycle time. The storage requirements of Berti, whose breakdown per structure is provided in Table I, is just 2.55 KB.

IV. Evaluation

A. Simulation Methodology

We use the recently modified version of ChampSim [9], a trace-driven simulator used for the 2nd and 3rd Data Prefetching Championships (DPC-2 [2] and DPC-3 [7]). Recent prefetching proposals [13], [17], [40], [48] are also coded and evaluated on ChampSim. The recently modified ChampSim extends the one provided with the DPC-3 with a decoupled front-end [45] and a detailed memory hierarchy support for address translation that further improve the baseline performance. Caches are non-inclusive, although Berti can work similarly with exclusion policies just by bypassing the allocation of memory blocks at the LLC. We faithfully model DRAM, including the queuing delays that contributes to the variable access time because of close vs. open page, page hit vs miss, DRAM bank conflicts, etc. Table II summarizes our system configuration, mimicking an Intel Sunny Cove microarchitecture [5], [6], [23].

We evaluate Berti with single-core and multi-core simulations. We warm-up the caches for 50M sim-point in-

structions [49] and collect statistics for the next 200M sim-point instructions. For multi-core simulations, we use heterogeneous mixes of single-threaded traces. For each mix, when a core finishes its 200M instructions, it gets replayed until all the cores finish their respective 200M instructions. For both single- and multi-core, we report performance in terms of IPC improvement (speedup) with respect to an L1D with an IP-stride prefetcher. We use the geometric mean to average the speedups obtained by the different single-thread traces.

Energy model. We also report the dynamic energy consumption of the memory hierarchy. We obtain the energy consumption of reads and writes to tag and data arrays at each cache level and DRAM with CACTI-P [37] and Micron DRAM power calculator [3]. Then, we compute the total energy expenditure by accounting for the number of accesses of each type across the memory hierarchy. We use 22 nm process technology for our energy calculations.

Workloads. We use traces from SPEC CPU2017 [55] and single-threaded GAP benchmarks [14]. We limit our study to memory-intensive traces (MemInt), i.e., those that showed at least one miss per kilo-instruction (MPKI) at the LLC in our modeled baseline system. All GAP traces (20) and 44 SPEC CPU2017 traces are memory-intensive.

SPEC CPU2017 traces were generated with the reference inputs. Both real (Twitter, Web, Road) and synthetic (Kron, Urand) graphs were used as input for the GAP benchmarks.

We also report performance for the CloudSuite benchmarks [22]. All traces are publicly available [4], [8], [10]. For multi-core experiments, we simulate 200 random heterogeneous mixes from SPEC CPU2017 and GAP.

Berti and variable cache fill latency. Modern memory hierarchies can have variable cache fill latency that comes from sources like MSHR contentions at the private and shared caches, read queue (RQ) and write queue (WQ) contentions at various levels of caches and DRAM controller. At the DRAM level, memory access time gets affected because of row buffer conflicts, bank conflicts, etc. Our simulator reflects all this variability. One of the primary reasons we propose Berti is because of the variable response latency. Even non uniform cache access (NUCA) LLCs with multiple banks can cause variable fill latency. For example, suppose in a NUCA LLC with two banks an IP sees local deltas of +1 and +2 that get mapped to bank-1 and bank-0, respectively, and the latency to bank-0 is different from bank-1. Even in this case, Berti is able to learn the best deltas while looking at the history, facilitating timely and accurate prefetching. Note that in our experiments, the fill latency ranges from 22 to 2098 cycles with an average of 278 cycles averaged across SPEC CPU2017, GAP, and CloudSuite benchmarks and multicore mixes.

Evaluated Prefetching Techniques. We compare the effectiveness of Berti with high performing L1D and L1D+L2 prefetchers. As Berti is an L1D prefetcher, we first compare its performance with prefetchers designed for L1D (no prefetching at the L2), and then with multi-level prefetching combinations. The L1D prefetchers are i) MLOP [48] (DPC-3, 3rd place), an extension of the BOP (DPC-2 winner), and ii) IPCP (DPC-3 winner published at ISCA 2020 [40]). For multi-level prefetching, we evaluate two state-of-the-art L2 prefetchers along with MLOP and Berti at the L1D: SPP-PPF [17], [35] and Bingo [13]. We also compare with a multi-level IPCP that uses IPCP both at the L1D and L2. The evaluated prefetchers have been briefly described in Section II-A. For all prefetchers, we use a highly tuned implementation as provided by the authors and tune it again for the parameters mentioned in Table II. Fine tuning was an easy exercise as all the competing prefetchers use ChampSim for their evaluation. Table III shows the configurations used for all the evaluated prefetchers.

Table II
SIMULATION PARAMETERS OF THE BASELINE SYSTEM.

Core	Out-of-order, hashed perceptron branch predictor [32], 4 GHz with 6-issue width, 4-retire width, 352-entry ROB
TLBs	L1 iTLB/dTLB: 64 entries, 4-way, 1 cycle STLB: 2048 entries, 16-way, 8 cycles
MMU Caches	2-entry PSCL5, 4-entry PSCL4, 8-entry PSCL3, 32-entry PSCL2, searched in parallel, one cycle
L1I	32 KB, 8-way, 4 cycles
L1D	48 KB, 12-way, 5 cycles, with a 24-entry, fully associative IP-stride prefetcher [18]
L2	512 KB 8-way associative, 10 cycles, SRRIP [31], non-inclusive
LLC	2 MB/core, 16-way, 20 cycles, DRRIP [31], non-inclusive
MSHRs	8/16/32 at L1I/L1D/L2, 64/core at the LLC
DRAM controller	One channel/4-cores, 6400 MTPS [19], FR-FCFS, 64-entry RQ and WQ, reads prioritized over writes, write watermark: 7/8th
DRAM chip	4 KB row-buffer per bank, open page, burst length 16, t_{RP}: 12.5 ns, t_{RCD}: 12.5 ns, t_{CAS}: 12.5 ns

Table III
CONFIGURATIONS OF EVALUATED PREFETCHERS.

SPP-PPF [17]	256-entry ST, 512-entry 4-way PT, 8-entry GHR, Perceptron weights with the following entries: 4096×4, 2048×2, 1024×2, and 128×1 entries, 1024-entry prefetch table, 1024-entry reject table
Bingo [13]	2 KB region, 64/128/4K-entry FT/AT/PHT
MLOP [48]	128-entry AMT, 500-update, 16-degree
IPCP [40]	128-entry IP table, 8-entry RST table, and 128-entry CSPT table

B. Speedup vs. storage requirements

Figure 7 summarizes the speedup of the evaluated prefetchers with respect to IP-stride for SPEC CPU2017 and GAP, along with their storage requirements. L1D prefetchers are shown with a circle, L2 prefetchers with a square, and multi-level (L1D+L2) prefetchers with a diamond.

Among the L1D prefetchers, Berti achieves the highest speedup with a size similar to IPCP, the prefetcher with the lowest storage budget. With only 2.55 KB of storage overhead, Berti improves performance by 8.5% over IP-stride

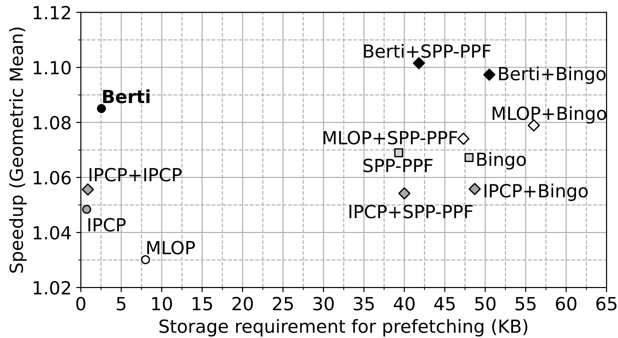


Figure 7. Speedup vs. storage requirements. Speedup is normalized to L1D IP-stride and averaged across memory-intensive SPEC CPU2017 and GAP traces. X+Y denotes prefetcher X at L1D and prefetcher Y at L2.

and 3.5% over IPCP, the state-of-the-art L1D prefetcher.

The Berti+SPP-PPF multi-level prefetcher obtains the highest speedup (10.2%, additional 1.5% on top of Berti at L1D) among all multi-level combinations with 41.8 KB combined storage for L1D and L2 prefetchers. However, the highlight of Figure 7 is that Berti at L1D without any prefetching at L2 outperforms all the multi-level prefetching combinations that do not include Berti.

C. Performance of Berti as an L1D Prefetcher

Figure 8 shows the speedup achieved by the L1D prefetchers for SPEC CPU2017 and GAP. Berti is the best prefetcher across both suites. On average, Berti at the L1D improves performance by 11.6% and 1.9% for SPEC CPU2017 and GAP, respectively. All three prefetchers achieve good speedups for SPEC CPU2017, and Berti outperforms IPCP and MLOP by 2.8% and 3.0%, respectively. The speedup differences are more significant with the GAP traces, where Berti is the only L1D prefetcher that improves IP-Stride, by up to 1.9%, while IPCP and MLOP are 2.9% and 7.8% below, respectively. Overall, across SPEC CPU2017 and GAP traces, Berti outperforms IP-stride and IPCP by 8.5% and 3.5%, respectively. This is significant performance improvement on top of the high-performing state-of-the-art IPCP prefetcher.

Figure 9 shows the individual speedup for the memory-intensive SPEC CPU2017 and GAP traces. For CPU2017, Berti achieves similar or significantly better results than the other prefetchers on all traces except for `CactuBSSN`. In this benchmark, we observe that the memory access instructions follow stride patterns. However, there are hundreds of these instructions executing interleaved. Therefore, to track the local behavior of instructions in this benchmark, Berti would need very large history and delta tables. In contrast, prefetchers that detect patterns in the global address stream do not have this problem, as is the case with MLOP or the IPCP GS class. Barring the exception of `CactuBSSN` where global deltas perform better than local deltas, Berti shows that local deltas are prevalent across a large number of benchmarks and it accurately selects them.

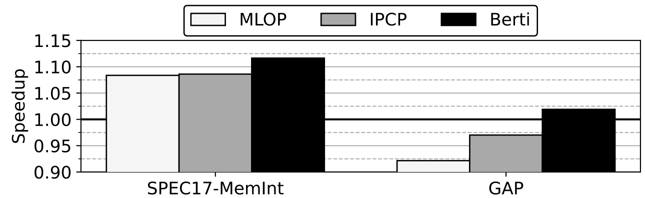


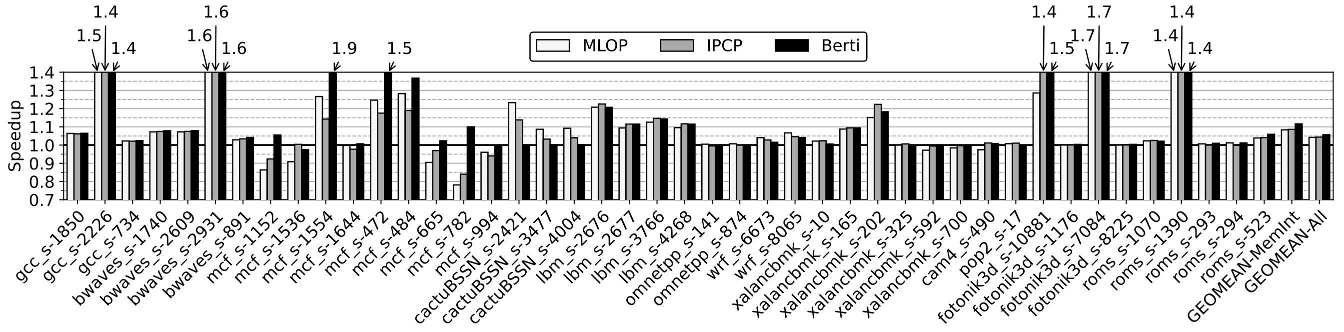
Figure 8. Speedup of L1D prefetchers compared to a system with L1D IP-stride for memory-intensive SPEC CPU2017 and GAP traces.

A key observation is that the state-of-the-art prefetchers do not consistently improve performance over IP-stride across all workloads, but Berti only shows a small degradation of 2.6% with respect to IP-stride for `mcf_s-1536`.

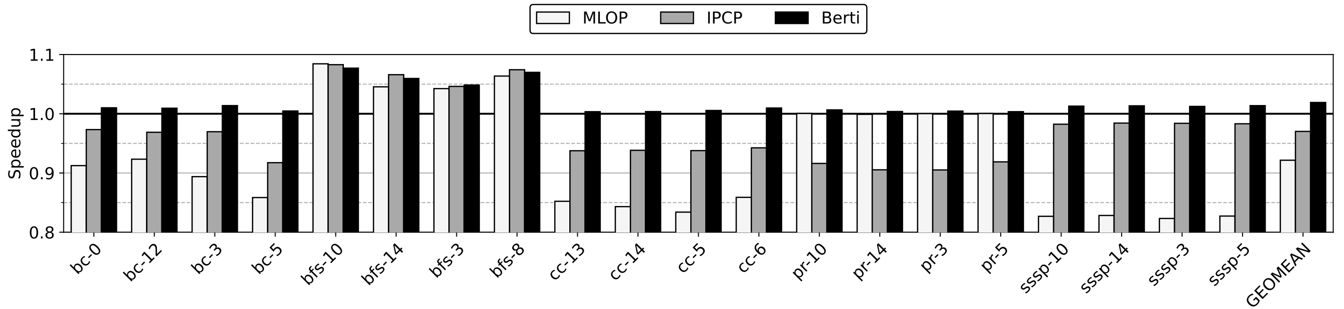
For SPEC CPU2017, Berti achieves its best result for `mcf_s-1554` where it provides speedups of 1.89×, 1.65×, and 1.49× with respect to IP-stride, IPCP, and MLOP, respectively. MLOP and IPCP achieve performance at least 1% below IP-stride on five and eight traces out of 44, the worst case being in `mcf_s-782` with drops of 16.0% and 21.9%, respectively. In `mcf_s-782`, only three IPs (`0x4049de`, `0x4049e5`, and `0x4049cc`) represent the 75% of all L1D accesses. MLOP uses a global delta to prefetch that is affected by the interleaving of accesses from these three IPs. IPCP uses the CS and CPLX class prefetchers but with an accuracy below 25%.

As for GAP, Berti is the best prefetcher for all the benchmarks but three (`bfs-8`, `bfs-10` and `bfs-14`). It consistently achieves similar or better results than IP-stride for all traces, while MLOP and IPCP perform worse than IP-stride in 12 and 16 benchmarks, respectively. In some cases, the MLOP slowdown is very significant, for example, 17.7% in `sssp-3`. We have also analyzed the behavior of the prefetchers in one of the GAP applications, namely `bc-5`. All `bc-5` IPs show a rather chaotic memory access pattern except for one that is very regular. IP-stride and Berti, by separately tracing the IPs, detect the regular IP pattern and prefetch correctly for it. They do not prefetch for the other IPs. MLOP fails due to the use of a global delta. The accesses issued by IPs with irregular pattern prevent discovering a global delta and therefore the prefetcher issues very few requests, and is not able to prefetch correctly for the regular IP. IPCP detects the delta pattern for the regular IP through its CPLX component, and prefetches correctly for it. However, the GS component generates many useless prefetches that drastically decreases the accuracy of IPCP and results in the loss of performance shown in Figure 9.

Accuracy. Figure 10 shows the accuracy of the L1D prefetchers. Berti is a very accurate prefetcher. On average, about 87.2% of its prefetched lines are useful compared to 62.4% for MLOP and 50.6% for IPCP. The effectiveness of IPCP is driven by the performance of several tiny prefetchers: a global stream prefetcher (GS class), a constant stride



(a) Memory-intensive SPEC CPU2017



(b) GAP

Figure 9. Speedup with Berti as an L1D prefetcher for (a) 44 SPEC CPU2017 and (b) 20 GAP memory-intensive traces normalized to L1D IP-stride. Geomean-all corresponds to the geometric mean of all the 95 SPEC CPU2017 traces.

prefetcher (CS class), and a complex stride prefetcher (CPLX class) that work in tandem. For regular access patterns, the CS prefetcher provides high accuracy. However, for complex access patterns, the effectiveness of the CPLX prefetcher is low, with an accuracy of 52.7% and 9.8% for SPEC CPU2017 [55] and GAP [14] workloads, respectively.

MLOP, like Berti, is based on the detection of the best timely deltas. However, it achieves much lower accuracy. The improvement of Berti over MLOP is mainly due to two factors: i) MLOP uses global deltas for the whole application while Berti detects different deltas for each IP. As we have shown in Section II-B, benchmarks such as `mcf` generate different delta patterns for each IP. ii) Berti uses a stringent policy to decide which deltas to use for issuing prefetch requests into L1D, as we have described in Sections III-B and III-C, while MLOP generates prefetch requests for the best delta with each lookahead regardless of its confidence.

Timeliness. The darker part of each bar in Figure 10 represents the prefetch requests whose retrieved data arrive late to L1D. Almost all prefetch requests generated by Berti are timely, while MLOP and IPCP produce a significant number of late requests. IPCP does not use any mechanism to adapt the prefetch requests timing to the miss latency, while MLOP and Berti do. However, Berti achieves better timeliness than MLOP due to specific and timely deltas for each IP.

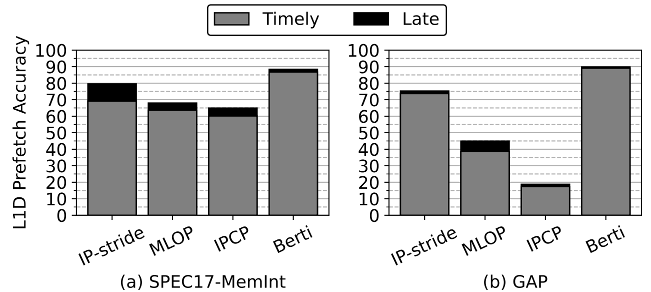


Figure 10. Prefetch accuracy at the L1D. Percentages of useful requests are broken down into timely (gray) and late (black) prefetch requests.

Coverage. Figure 11 shows demand misses per kilo instructions (MPKI) at the L1D, L2, and LLC with and without L1D prefetchers. Berti and IPCP achieve a similar reduction of misses in L1D (8.7% in GAP, 33.4% in SPEC CPU2017) and slightly higher than MLOP. However, Berti manages to eliminate more misses than IPCP and MLOP at L2 and LLC due to its line preloading policy directed by the L1D prefetcher. The biggest differences are observed in GAP, where Berti reduces LLC demand misses by 17.7% and 12.4% compared to MLOP and IPCP, respectively. Similarly, Berti reduces L2 demand misses by 6.7% and 5.6% compared to MLOP and IPCP, respectively.

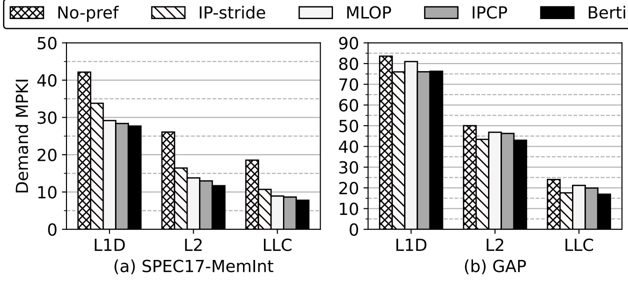


Figure 11. Prefetch coverage in terms of average L1D, L2, and LLC demand MPKIs for all L1D prefetchers.

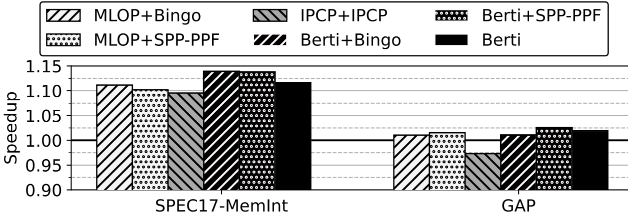


Figure 12. Spedup with multi-level prefetching normalized to L1D IP-stride.

D. Multi-level prefetching performance

Figure 12 shows the speedup achieved with the multi-level prefetching combinations compared to a system with IP-stride. We select the best five multi-level prefetching combinations out of all possible combinations of L1D and L2 prefetchers. Multi-level prefetching combinations do not offer a significant performance boost. The best multi-level prefetching combinations without Berti are MLOP+Bingo for SPEC CPU2017 and MLOP+SPP-PPF for GAP. In both cases, these combinations achieve a similar speedup to Berti alone, with a storage requirement 22 and 18 times higher, respectively. IPCP at L1D and L2 (IPCP+IPCP), with a meager hardware budget as Berti, achieves a significantly lower speedup than Berti alone at the L1D, especially in GAP, with a difference of 4.6%.

Adding a prefetcher at the L2 cache along with Berti at the L1D achieves a moderate performance gain. The most significant gain is 2.0% and is obtained with the Berti+Bingo configuration in the memory-intensive subset of SPEC CPU2017 traces. Given the high hardware cost of the L2 prefetchers, the configuration with Berti alone at the L1D seems to be a better design in terms of performance and storage trade-off.

Coverage. Figure 13 shows demand MPKIs at the L1D, L2, and LLC for the multi-level prefetching combinations. We also show MPKIs without prefetching at L2 for ease of analysis. MLOP+Bingo and MLOP+SPP-PPF decrease MPKI relative to MLOP alone in both L2 and LLC consistently across all suites (maximum reduction in MPKI from 13.8 to 11.7 at L2 for SPEC CPU2017). Adding a prefetcher at L2 is less effective for IPCP and Berti. In both cases, MPKIs at L2 and LLC decrease for SPEC CPU2017 but remain the same or even increase slightly when working with the irregular access patterns of GAP. As a result, the MPKIs at the L2 and LLC achieved by Berti at the L1D are always better than those obtained by multi-level prefetchers with no Berti, except for MLOP+Bingo in SPEC CPU2017.

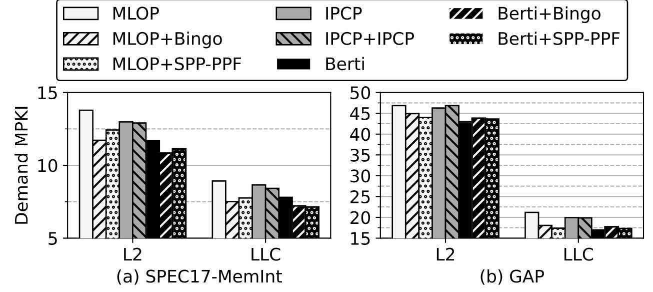
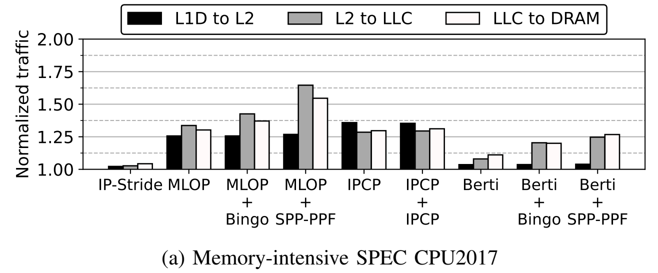
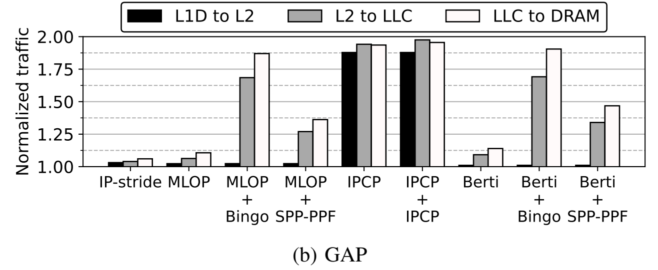


Figure 13. Prefetch coverage in terms of average L2 and LLC demand MPKIs with multi-level prefetching.



(a) Memory-intensive SPEC CPU2017



(b) GAP

Figure 14. L2, LLC and DRAM demand and prefetch traffic normalized to no-prefetching.

E. Memory hierarchy traffic and energy

Figure 14 shows the traffic between the different levels of the memory hierarchy (demand and prefetch requests) for different prefetching combinations normalized to no prefetching. All prefetchers increase traffic as a result of the useless blocks they request. For the systems with prefetcher only at the L1D, we can observe how the traffic increase is inversely proportional to the prefetcher accuracy for SPEC CPU2017 (refer Figure 10). Consequently, Berti is the prefetcher with the lowest traffic increase at all levels. For GAP, MLOP increases traffic marginally, despite its low accuracy, because it detects few patterns and generates scarce prefetch requests. Berti increases traffic with L2, LLC and

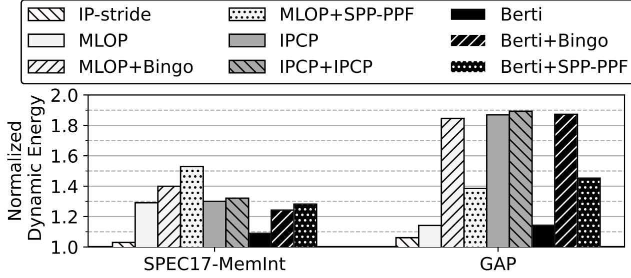


Figure 15. Dynamic energy consumption in the memory hierarchy normalized to no-prefetching.

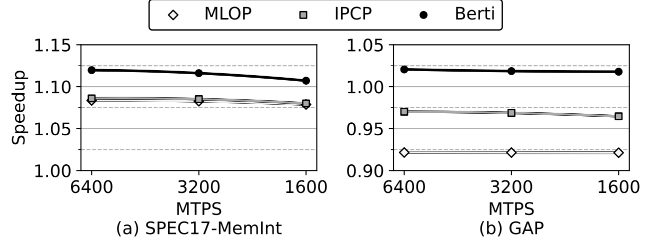


Figure 16. Performance of L1D prefetchers in constrained DRAM bandwidth, in MTPS.

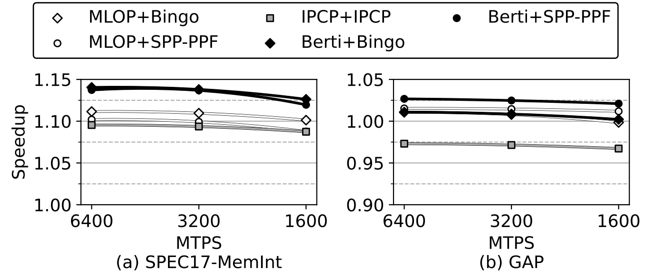


Figure 17. Performance of multi-level prefetching in constrained DRAM bandwidth, in MTPS.

DRAM by 1.0%, 9.2% and 13.9% respectively, whereas IPCP increases traffic at these three levels around 90%.

The L2 prefetchers Bingo and SPP-PPF added to MLOP and Berti on L1 significantly increase traffic with LLC and DRAM, especially at GAP due to the irregular access patterns. MLOP+Bingo induces 69.0% additional off-chip traffic compared to MLOP alone, while Berti+Bingo adds 67.2% additional off-chip traffic compared to Berti alone.

Energy efficiency. Figure 15 shows the average dynamic energy consumption in the memory hierarchy (L1D, L2, LLC, and DRAM) normalized to no prefetching. As expected, there is a direct correlation between traffic and energy consumption overheads in the memory hierarchy. If we focus on the state-of-the-art L1D prefetchers, Berti consumes the least extra energy for SPEC CPU2017 (9.0% vs. 29.1 and 30.1% for MLOP and IPCP), despite achieving the highest speedup (see Figure 8). As for GAP, the energy overheads of Berti and MLOP are similar (14.3% vs. 14.2%), and significantly lower than for IPCP (86.9%). Berti is the only prefetcher that manages to translate its dynamic energy increase into speedup. The L2 Bingo and SPP-PPF prefetchers on top of MLOP and Berti significantly increase energy consumption, especially in the case of Bingo for GAP, with increases of over 60% with respect to MLOP and Berti alone.

F. Effect of constrained DRAM bandwidth

So far, we have considered the latest DDR5-6400 channel per four cores that provides 6400 million transfers per second (MTPS) with a per-core DRAM bandwidth of approx. 12.8 GBps. This Section evaluates prefetchers with DRAM bandwidth configurations such as DDR4-3200 (MTPS of 3200) and DDR3-1600 (MTPS of 1600) [19]. Figures 16 and 17 show the effect of DRAM bandwidth on speedup for L1D and multi-level prefetching, respectively. When moving from 6400 to 1600 MTPS, the performance loss is negligible for all the prefetchers with GAP traces and moderate for SPEC CPU2017 traces (maximum reduction of 4.1% with Berti and Berti+SPP-PPF).

G. CloudSuite performance

Figure 18 shows speedup with the CloudSuite benchmarks for L1D and multi-level prefetching combinations. `Classification` is one benchmark where all the prefetchers fail except Berti, thanks to its high prefetch accuracy. Note that for some of the benchmarks like `cloud9` and `nutch`, even an ideal L1D prefetcher (L1D with a hit rate of 100%) fails to provide significant performance, which shows that there is limited scope for data prefetching. The primary reason for this trend is that the L1D MPKI of CloudSuite without prefetch is low: 6.9 on average, with a maximum of 14.5, while the average L1D MPKI of SPEC and GAP is 42.2 and 83.6, respectively. On the other hand, the L1I MPKI of CloudSuite traces are higher than SPEC and GAP traces.

H. Interaction with a temporal prefetcher

We simulate managed irregular stream buffer (MISB) prefetcher [59], a storage efficient version of ISB [30] at L2 with MLOP, IPCP, and Berti at L1D, as shown in Figure 19. ISB is an address correlation-based data prefetcher that correlates cache accesses at a new indirection level named structural address space. Berti with MISB improves the effectiveness of multi-level prefetching for CloudSuite traces, in particular for `Cassandra` and `Classification`. For SPEC CPU2017 and GAP, MISB performs worse than SPP-PPF with MLOP and Berti at the L1D. Note that the performance improvement with CloudSuite comes with a storage overhead of 98 KB with MISB, out of which 32 KB is used for the metadata cache and 17 KB for the Bloom filter.

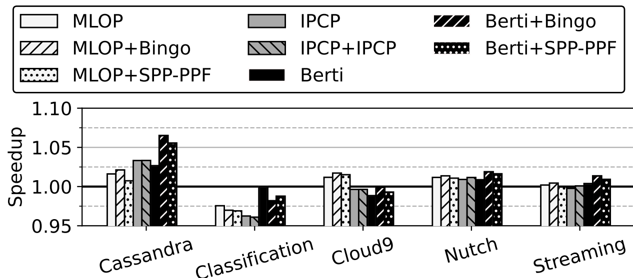


Figure 18. Speedup for CloudSuite.

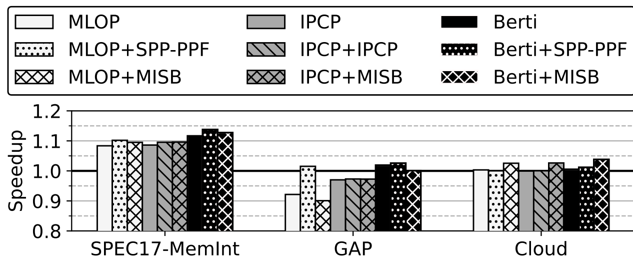


Figure 19. Speedup with and without MISB.

I. Multi-core performance

Figure 20 shows speedup on a 4-core simulated system averaged across 200 randomly generated heterogeneous mixes based on memory-intensive SPEC CPU2017 and GAP traces. Among the L1D prefetchers, Berti performs the best with a performance improvement of 16.2%, outperforming both MLOP and IPCP on average. There are only nine mixes in which MLOP and/or IPCP gain more than 10% over Berti, and `CactuBSSN` is part of seven of these mixes. Berti outperforms competing prefetchers for majority of the mixes that do not have `CactuBSSN`. Overall, Berti performs better because in the case of multicore systems, per core available DRAM bandwidth goes down because of cross-core contention. Thanks to Berti's timely and accurate deltas, it is still able to deliver high coverage even in the presence of shared DRAM bandwidth contention.

Berti at L1D also outperforms other multi-level prefetching combinations making a strong case for Berti as an L1D-only prefetcher. Note that Berti outperforms MLOP+Bingo, the combination of the second place and first place prefetchers in the 4-core evaluations at the DPC-3.

J. Sensitivity to design choices

Effect of L1 and L2 watermarks. Figure 21 shows the effect of L1 and L2 confidence watermarks on overall speedup averaged across single-core SPEC CPU2017 and GAP benchmarks, normalized to the baseline system. Our chosen watermarks, more than 65% for L1 and in between 35% to 65% for L2 provide the sweet-spot in terms of prefetch accuracy and prefetch coverage. Usage of extremely small or extremely large watermarks affect both coverage and accuracy, and negatively affects speedup. Interestingly, a large

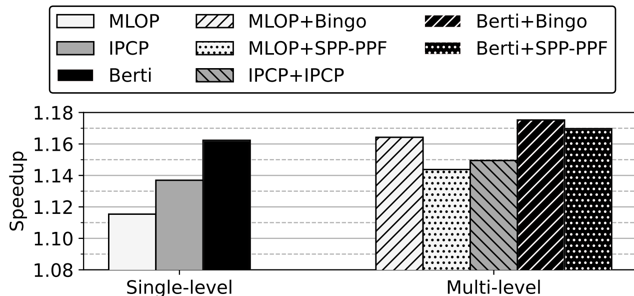


Figure 20. Summary of multi-core speedups relative to a system with L1D IP-stride prefetcher.

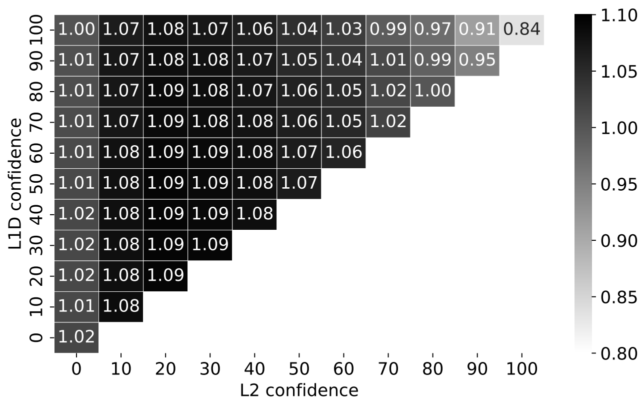


Figure 21. Normalized speedup with different L1D and L2 confidence watermarks averaged across memory intensive SPEC CPU2017 and GAP benchmarks. Speedup is rounded to two decimal places (1.085 is rounded to 1.09).

number of watermark configurations provide similar benefit in terms of speedup. Our chosen high watermarks provide maximum speedup with the maximum prefetch accuracy.

Effect of the size of Berti tables. Figure 22 shows the effect of the size of the Berti tables (history table, table of deltas, and the number of deltas) on speedup. Decreasing the size of the table of deltas by a quarter degrades performance by 12.1%, whereas decreasing the number of deltas by a quarter reduces performance by 1.2%. Also, doubling/quadrupling the size of the tables provides a marginal performance gain. `CactuBSSN` is one outlier where increasing the table sizes to 1024 entries with 1024 sets improve performance by 22%.

Effect of the latency counter. In our evaluations, we use a 12-bit latency counter per line at the L1D. When we increase its size to 32 bits, performance is not improved. However, using a small 4-bit timestamp, we see a performance drop from 1.16 to 1.07, and from 1.02 to 0.98, for SPEC CPU2017 and GAP, respectively.

Effect of cross-page prefetching. As Berti is an L1D prefetcher and operates on virtual addresses, it does cross-page prefetching as long as prefetch requests get a hit in the STLB. To understand the utility of cross-page prefetching,

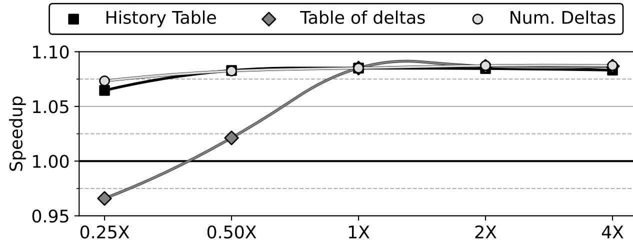


Figure 22. Speedup vs. size of Berti tables and number of deltas. 0.25× to 4× correspond to one-fourth and four times, respectively.

we evaluate Berti, where we do not issue prefetch requests (but keep training) that cross an OS page. We see an average performance drop from 1.02 to 1.01 and 1.16 to 1.10 for GAP and SPEC CPU2017 traces, respectively. The performance drop shows that most of the deltas selected by Berti are within the OS page boundary of 4 KB.

V. Related Work

In Section IV we presented a quantitative comparison of Berti with recent hardware prefetching techniques [13], [17], [35], [38], [40], [48]. In this Section we compare other relevant prefetching techniques qualitatively.

Temporal prefetchers. Temporal prefetchers track the temporal order of cache-line accesses (and not the deltas) [12], [27], [30], [33], [52], [57]. Temporal prefetchers usually demand hundreds of KBs of storage, which demands the storage of prefetch metadata in the off-chip memory. Some of the recent works on temporal prefetching are in the pursuit of improving the storage overhead without affecting the prefetch coverage [58], [59]. Berti, on the other hand, incurs a storage overhead of just 2.55 KB per core.

Spatial prefetchers. Compared to temporal prefetchers, spatial prefetchers are lightweight in terms of storage overhead and usually learn memory access patterns within a small spatial region of a few KBs. Conventional prefetchers like stride [20] and stream [24], [28], [53] are already deployed on commercial processors. Timely Stride prefetching improves the timeliness of conventional stride prefetchers [60]. However, it does not provide better prefetch coverage when compared with state-of-the-art L1D and L2 prefetching techniques. Spatial prefetchers like Spatial Memory Streaming (SMS) [53] (similar to Bingo) usually learn single repeating deltas or bit patterns within a spatial region, where a set bit denotes a cache line that should be prefetched. All these techniques do not consider prefetch timeliness.

Kill the program counter (KPC) proposes a holistic cache replacement and prefetching framework [36]. However, the prefetching technique is similar to SPP, with similar performance improvements as SPP. Multi-level adaptive prefetching based on performance gradient tracking [44] (3rd place in DPC-1 [1]) is one of the first proposals that propose a correlation between an IP and delta sequences. DSPatch [16] tunes a hardware prefetcher based on available DRAM bandwidth and selects memory access patterns based on prefetch accuracy (if the available DRAM bandwidth is low) and prefetch coverage (if the available DRAM bandwidth is high). Overall, SPP-PPF performs marginally better than SPP+DSPatch.

Machine learning for hardware prefetching. Machine learning (ML) has been used for microarchitecture research, and ML techniques for data prefetching have been proposed in recent years [15], [25], [51]. In ISCA 2021, a prefetching competition with ML techniques shows that non-ML techniques still outperform with limited storage. However, ML techniques have the potential to learn highly complex memory access patterns, and Pythia [15] shows that with a high performing L2 prefetcher. Berti is an L1D prefetcher in contrast to Pythia, and with Berti at the L1D, we find negligible performance improvement with Pythia (less than 1%).

Prefetch filters and throttling mechanisms. Similar to PPF [17] and DSPatch [16], there are proposals that control the aggressiveness of prefetchers by controlling its prefetch degree and distance, or decides whether to prefetch into the L2 or to the LLC [11], [21], [26], [41], [42], [43], [54]. These techniques incur additional storage and perform well for conventional prefetchers with low prefetch accuracy. However, with Berti, the accuracy is significantly higher than prior prefetching techniques, and the implicit confidence mechanism acts like a prefetch throttler.

VI. Conclusions

We proposed Berti and made a case for an L1D prefetcher based on local, timely deltas. Berti learns the best delta to prefetch, keeping timeliness (in the form of time to prefetch an address) and prefetch accuracy in mind. We showed that Berti could learn varieties of memory access patterns. We quantified the effectiveness of Berti across SPEC CPU2017 and GAP workloads, and showed high prefetch accuracy and timely prefetching into the cache hierarchy. On average, Berti outperforms state-of-the-art L1D and L2 prefetchers. Berti is equally effective even in the constrained DRAM bandwidth scenarios and also for multi-core mixes. Berti consumes the least dynamic energy at the memory hierarchy among all state-of-the-art prefetchers. In summary, Berti provides high prefetch accuracy, timely prefetching, and good coverage with a limited storage overhead of 2.55 KB per core.

Acknowledgements

This work was supported by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe" (grants PID2019-105660RB-C21, RTI2018-098156-B-C53), the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 819134), and by Government of Aragon (T5820R research group).

APPENDIX

A. Abstract

This artifact contains all the information necessary to reproduce the main experiments in the paper. We describe how the required software and the elements that compose it can be obtained, and how to run the artifact.

B. Artifact Check-list & Meta-information

- **Program:** ChampSim
- **Compilation:** GNU GCC 7.5.0
- **Data set:** SPEC CPU2017 traces from 3rd Data Prefetching Championship (http://hpca23.cse.tamu.edu/champsim-traces/speccpu/)
- **Run-time environment:** an AMD64 processor
- **Hardware:** tested on an AMD EPYC 7702P
- **Metrics:** speedup and L1D Accuracy
- **Output:** three PDF files with graphs, single-thread memory-intensive SPEC CPU2017 speedup and L1D accuracy
- **How much disk space is required?:** approximately 22 GB
- **How much time is needed to prepare the workflow?:** approximately 30 minutes
- **How much time is needed to complete experiments?:** approximately 1 hour 30 minutes (running in parallel, using 127 threads)
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** GPL-3.0
- **Archived (provide DOI)?:** 10.5281/zenodo.6921331

C. Description

1) How to access: The software can be obtained from GitHub: https://github.com/agusnt/Berti-Artifact

Use the following command to clone the repository:

```
$ git clone
https://github.com/agusnt/Berti-Artifact
```

2) Software Dependencies: We test the artifact on a system with these features:

- Ubuntu 18.04.6 LTS
- Linux Kernel 5.4.0
- Python 3.6.9
- Bash 4.4.20
- GCC 7.5.0
- Curl 7.61
- **Python3 Packages**
  - matplotlib 3.3.4
  - pprint 0.1
  - numpy 1.19.1
  - scipy 1.5.4

However, in our tests, new GNU/Linux systems were able to run the artifact. Only newer GCC compilers may raise errors in execution. In order to ease the process of running the artifact, we provide two options in the main script: (1) The `-g` flag downloads and builds GCC 7.5.0 from scratch, and (2) `-d` uses Docker to build the simulator. The Python3 Packages can be installed using pip3.

3) Data sets: The traces for the full set of experiments were downloaded from different championships. However, for this artifact only the SPEC CPU2017 traces from the 3rd Data Prefetching Championship (http://hpca23.cse.tamu.edu/champsim-traces/speccpu/) are needed. These traces are automatically downloaded by the artifact. GAP (from ML for Computer Architecture and Systems https://sites.google.com/view/mlarchsys/) and CloudSuite (from the 2nd Cache Replacement Championship https://crc2.ece.tamu.edu) are also used in the paper, so all results in the paper are easily reproducible by updating our scrips.

D. Installation & Experimental workflow

The artifact is ready to be built and run automatically by executing the `run.sh` script. The overall flow is as follows:

1) Clone the artifact:
   ```
   $ git clone
   https://github.com/agusnt/Berti-Artifact
   ```
2) Enter the cloned repository:
   ```
   $ cd Berti-Artifact
   ```
3) Run the script:
   ```
   $ ./run.sh
   ```
   Optionally, you can speed up the simulations by running them in parallel:
   ```
   $ ./run.sh -p [number of threads]
   ```
   In case of an error while building the artifact or running the simulations, try compiling instead with the Docker flag; it can be used along with the parallelization option (`-p`):
   ```
   $ ./run.sh -d
   ```

E. Evaluation and expected results

The artifact provides speedup and L1D accuracy results for the memory-intensive single-thread SPEC CPU2017 traces at the end of the simulation. The execution of the artifact prints the following information:

```
  Building with Docker
  Running in Parallel
  Download SPEC CPU2017 traces [44/44]
  Building Berti... done
  Building MLOP... done
  Building IPCP... done
  Building IP Stride... done
  Making everything ready to run... done
  Running... done
  Results, it requires numpy, scipy,
maptlotlib and pprint
  Parsing data... done
  SPEC CPU2017 Memory Intensive SpeedUp
```

Prefetch	Speedup	L1D Accuracy
IPCP	09%	64.9%
MLOP	08%	68.0%
Berti	12%	88.0%

```
  Generating Figure 8
SPEC_CPU2017-MemInt... done
  Generating Figure 9(a)... done
  Generating Figure 10
SPEC_CPU2017-MemInt... done
  Removing Temporal Files... done
```

It generates three graphs in PDF format named `fig8.pdf`, `fig9.pdf` and `fig10.pdf`, in the working directory (where `run.sh` is placed).

F. Experiment customization

We implement two options to run the artifact in system with newer GCC compilers: (1) `./run.sh -g` downloads and builds GCC 7.5.0 from scratch to build the artifact, and (2) `./run.sh -d` builds the artifact with Docker.

To speedup the artifact we provide a `-p [number of threads]` flag that allows to run the experiments in parallel.

The remaining parameters of our script are: `-v` verbose mode, `-h` a help menu, `-c` deletes all temporal files, i.e. traces.

G. Notes

The L1D accuracy reported by ChampSim is calculated differently from the reported by the artifact. Instead, the L1D accuracy reported by the artifact is calculated as:

$$\frac{L1DPrefetchLate + L1DPrefetchTimely}{L1DPrefetchFill}$$

Our L1D accuracy formula represents the unnecessary traffic generated by the prefetcher, i.e. an accuracy of 90% indicates that there is 10% of unnecessary traffic. The nominator includes all successful prefetches (timely or late), prefetches that have not caused unnecessary extra traffic, and the denominator represents the data brought into the cache by the prefetcher.

References

[1] "The 1st data prefetching championship (dpc-1)," Feb. 2009. [Online]. Available: https://jilp.org/dpc/

[2] "The 2nd data prefetching championship (dpc-2)," Jun. 2015. [Online]. Available: https://comparch-conf.gatech.edu/dpc2/

[3] "Micron dram power calculator," Dec. 2015. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf

[4] "Cloudsuite traces for champsim," Nov. 2017. [Online]. Available: https://www.dropbox.com/sh/pgmnzfr3hurlutq/AACciuebRwSAOzhJkmj5SEXBa/CRC2_trace?dl=0&subfolder_nav_tracking=1

[5] "SunnyCove microarhcitecture," May 2018. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

[6] "SunnyCove microarhcitecture latency," May 2018. [Online]. Available: https://www.7-cpu.com/cpu/Ice_Lake.html

[7] "The 3rd data prefetching championship (dpc-3)," Jun. 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/

[8] "SPEC CPU 2017 traces for champsim," Feb. 2019. [Online]. Available: https://hpca23.cse.tamu.edu/champsim-traces/speccpu/index.html

[9] "ChampSim simulator," May 2020. [Online]. Available: http://github.com/ChampSim/ChampSim

[10] "GAP traces for champsim," Mar. 2021. [Online]. Available: https://utexas.app.box.com/s/2k54kp8zvrqdfaa8cdhfquvcxwh7yn85/folder/132804598561

[11] J. Albericio, R. Gran, P. Ibáñez, V. Viñals, and J. M. Llabería, "Abs: A low-cost adaptive controller for prefetching in a banked shared last-level cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 19:1–19:20, Jan. 2012.

[12] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *24th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 131–142.

[13] 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, pp. 399–411.

[14] S. Beamer, K. Asanović, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, Aug. 2015.

[15] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *54th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 1121–1137.

[16] R. Bera, A. V. Nori, , O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *52nd Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2019, pp. 531–544.

[17] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–13.

[18] Y. Chen, L. Pei, and T. E. Carlson, "Leaking control flow information via the hardware prefetcher," *CoRR*, vol. abs/2109.00474, Sep. 2021.

[19] DDR, "DDR standards." [Online]. Available: https://en.wikipedia.org/wiki/Double_data_rate

[20] J. Doweck, “Inside intel core microarchitecture and smart memory access,” in *Intel whitepaper*, 2006.

[21] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, “Coordinated control of multiple prefetchers in multi-core systems,” in *42nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 316–326.

[22] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: A study of emerging scale-out workloads on modern hardware,” in *17th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2012, pp. 37–48.

[23] A. Fog, “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers,” 2020. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[24] B. Grayson, J. Rupley, G. D. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, “Evolution of the samsung exynos cpu microarchitecture,” in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 40–51.

[25] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, “Learning memory access patterns,” in *35th Int'l Conf. on Machine Learning (ICML)*, Jul. 2018, pp. 1924–1933.

[26] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, “Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors,” in *27th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Nov. 2018, pp. 28:1–28:11.

[27] Z. Hu, M. Martonosi, and S. Kaxiras, “Tcp: Tag correlating prefetchers,” in *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 317–326.

[28] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, “Effective stream-based and execution-based data prefetching,” in *18th Int'l Conf. on Supercomputing (ICS)*, Jun. 2004, pp. 1–11.

[29] A. Jain, “Exploiting long-term behavior for improved memory system performance,” Ph.D. dissertation, The University of Texas at Austin, May 2016.

[30] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” in *46th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 247–259.

[31] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” in *37th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2010, pp. 60–71.

[32] D. A. Jiménez and C. Lin, “Dynamic branch prediction with perceptrons,” in *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 197–206.

[33] D. Joseph and D. Grunwald, “Prefetching using markov predictors,” in *24th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1997, pp. 252–263.

[34] N. S. Kalani and B. Panda, “Instruction criticality based energy-efficient hardware data prefetching,” *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 146–149, 2021.

[35] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, “Path confidence based lookahead prefetching,” in *49th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 60:1–60:12.

[36] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, “Kill the program counter: Reconstructing program behavior in the processor cache hierarchy,” in *22nd Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 2017, pp. 737–749.

[37] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, “Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques,” in *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.

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

[39] C. Öztürk, I. B. Karsli, and R. Sendag, “An analysis of address and branch patterns with patternfinder,” in *Int'l Symp. on Workload Characterization (IISWC)*, Oct. 2014, pp. 232–242.

[40] S. Pakalapati and B. Panda, “Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching,” in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 118–131.

[41] B. Panda and S. Balachandran, “Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers,” *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 13–16, Jan. 2016.

[42] B. Panda, “SPAC: A synergistic prefetcher aggressiveness controller for multi-core systems,” *IEEE Transactions on Computers (TC)*, vol. 65, no. 12, pp. 3740–3753, Dec. 2016.

[43] B. Panda and S. Balachandran, “CAFFEINE: A utility-driven prefetcher aggressiveness engine for multicores,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, pp. 30:1–30:25, Aug. 2015.

[44] L. M. Ramos, J. L. Briz, P. E. Ibáñez, and V. Viñals, “Multi-level adaptive prefetching based on performance gradient tracking,” *The Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–14, Jan. 2011.

[45] G. Reinman, B. Calder, and T. Austin, “Fetch directed instruction prefetching,” in *32nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 16–27.

[46] A. Ros, “Berti: A per-page best-request-time delta prefetcher,” in *The 3rd Data Prefetching Championship*, Jun. 2019.

[47] A. Ros and A. Jimborean, “A cost-effective entangling prefetcher for instructions,” in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2021, pp. 99–111.

[48] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in *The 3rd Data Prefetching Championship*, Jun. 2019.

[49] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[50] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *48th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 141–152.

[51] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *26th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 2021, pp. 861–873.

[52] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 69–80.

[53] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 252–263.

[54] 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)*, Feb. 2007, pp. 63–74.

[55] Standard Performance Evaluation Corporation, "SPEC CPU2017," 2017. [Online]. Available: http://www.spec.org/cpu2017

[56] K. Sundaram and A. Radhakrishnan, "Address re-ordering mechanism for efficient pre-fetch training in an out-of-order processor," U.S. Patent 9542323B2, Sep. 2014.

[57] D. A. Varkey, B. Panda, and M. Mutyam, "RCTP: Region correlated temporal prefetcher," in *35th Int'l Conf. on Computer Design (ICCD)*, Nov. 2017, pp. 73–80.

[58] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *52nd Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2019, pp. 996–1008.

[59] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 449–461.

[60] H. Zhu, Y. Chen, and X.-H. Sun, "Timing local streams: improving timeliness in data prefetching," in *24th Int'l Conf. on Supercomputing (ICS)*, Jun. 2010, pp. 169–178.