Presented at the 4th Data Prefetching Championship (DPC4), held in conjunction with HPCA 2026

Global Berti: Simultaneous Streaming and Spatial Prefetching

Gilead Posluns
University of Toronto
Toronto, Ontario, Canada
gil.posluns@mail.utoronto.ca

Mark C. Jeffrey
University of Toronto
Toronto, Ontario, Canada
mcj@ece.utoronto.ca

ABSTRACT

Prefetchers predict future memory accesses to bring data into the cache before it is accessed by recognizing and repeating or extrapolating an application's memory access pattern. To date, no prefetcher predicts all of the disparate access patterns across all applications. The Berti prefetcher captures streaming access patterns where a *single* instruction accesses addresses a consistent delta apart from each other. It fails to capture spatial patterns where *different* instructions access addresses with a consistent delta. We have found that by accessing Berti's tables in two orientations, a prefetcher can detect and predict both streaming and spatial patterns, resulting in speedups of up to 1.89× over baseline Berti for almost zero additional hardware cost.

KEYWORDS

streaming data prefetchers, spatial data prefetchers

1 INTRODUCTION

Applications rely on hardware prefetchers to avoid losing performance to cache misses. Prefetches rely on the past to predict the future by learning application access patterns. Correlating prefetchers [1, 4, 11] record sequences of cache misses and replay them if they repeat, but do not generalize to new addresses. Spatial prefetchers [5, 7] also record a sequence of misses, but generalize their access pattern to replay the same sequence of accesses in a new region of memory (typically a page). Finally, offset-based prefetchers predict that a future access will occur a certain *delta* from a demand access. Offset-based predictors capture streaming access patterns, such as iterating through an array [3, 15, 17, 21], or capture a sequence of accesses to a data structure [6, 13, 14] similarly to spatial prefetchers, but not both.

The Berti [15] prefetcher is a recent design that measures memory access latency and leverages this information to determine how far ahead it should prefetch a streaming access pattern. Berti is designed to capture local patterns, with consistent offsets between accesses by the same instruction pointer (IP). This addresses the limited coverage of the Best-Offset Prefetcher [14] which found a single global offset based on latency information. Although Berti improves coverage over the Best-Offset Prefetcher, it does not measure access offsets between different IPs, which means that it misses spatial access patterns. Our insights are that *(i)* accessing Berti's per-IP history table by column instead of by row recovers enough global history information to find inter-IP deltas, and *(ii)* spatial patterns are only worth detecting and recording in the absence of streaming patterns, allowing us to reduce overheads. Based on these insights, we propose *Global Berti*, a prefetcher that detects both streaming and spatial patterns, resulting in speedups of up to 1.89× over baseline Berti for only 264B of additional storage overhead.

2 MOTIVATION

The code in Listing 1 exhibits both streaming and spatial access patterns. Fig. 1 shows how these access patterns are laid out in space and time. Line 3 strides across the list array: each of its instances accesses data a predictable offset from the last instance. This *streaming* access pattern is predictable by simple stride and stream prefetchers [3, 21], as well as by more sophisticated prefetchers like Berti [15]. In contrast, Line 7 accesses data (`h->next`) a constant offset from the data accessed by the preceding Line 6 (`h->data`). This is an example of a *spatial* access pattern, which is predictable by spatial prefetchers like Bingo [5] or Gaze [7], as well as by global offset prefetchers like Pythia [6]. Line 4 exhibits *both* streaming and spatial access patterns. Each access instance of Line 4 is both a consistent offset (`sizeof(ListNode*)`) from the previous Line 4 and a different consistent offset (`tails - heads`) from the previous Line 3. This will be the case any time multiple fields are accessed in an array of data structures, or any time multiple streaming access patterns with the same stride are interleaved. Any time a spatial access pattern includes an IP with a streaming access pattern, all IPs in the spatial pattern will have streaming patterns. Therefore, when a streaming access pattern is present, searching for spatial access patterns is redundant.

Our insight that spatial access patterns are only relevant when there are no streaming access patterns simplifies the process of searching for spatial patterns. Because the accesses of a spatial pattern have no streaming pattern, we cannot prefetch several iterations in the future. Therefore, only the current pattern instance matters, which means that we can confine our search for spatial patterns to only the most recent instance of each IP in the global history. Given a table mapping IPs to local address histories (as exists in streaming prefetchers, including Berti), looking at only the most recent entry for each IP provides a curated version of the global address history, including only the addresses that might be relevant for useful spatial patterns.

Organizing our address history per-IP, as streaming prefetchers do, actually makes it *easier* to find useful spatial patterns. We simply need to read a column from the table.

3 BACKGROUND ON BERTI

The key idea of the Berti [15] prefetcher is to find *timely deltas* for streaming access patterns. Timeliness is important to prefetchers because a prefetch issued too late will fail to hide all the latency for the access it predicted, whereas a prefetch issued too early may be uselessly evicted before being accessed, while potentially evicting still-useful data itself. The best distance to prefetch ahead in a streaming access pattern depends on how the data is being used. A loop that does 1000 cycles of processing on each value it reads (i.e., high operational intensity [20]) would be satisfied by prefetching only the next access, whereas a loop that merely

```
int sum = 0;
for (int i = 0; i < N_LISTS; i++) {
  ListNode* h = heads[i];
  ListNode* t = tails[i];
  while (h != t) {
    sum += h->data;
    h = h->next;
  }
}
f(sum);
```

Listing 1: Example code with streaming and spatial access patterns. Color highlights correspond to the patterns in Fig. 1

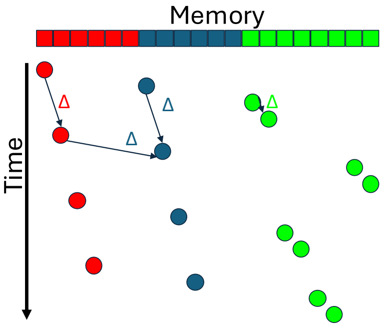

Figure 1: Illustrating the memory access patterns in space and time.

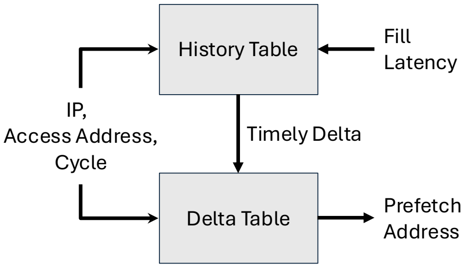

Figure 2: Berti [15] tables and their interfaces

sums up a large array (i.e., low operational intensity) could require prefetching hundreds of iterations ahead to hide all of its memory latency in steady state.

Prefetchers like Pythia [6] or the Micro-Armed Bandit [9] leverage reinforcement learning techniques to find an appropriate prefetch distance. Berti instead maintains a longer history per IP, looks back into it to find what distance it should have used to be timely (the timely delta), and uses that distance going forward. Timely deltas also resolve the issue of complex strides: patterns of accesses that are consistent, but not identical. For example, an instruction that accesses addresses 0x0, 0x4, 0xC, 0x10, 0x18, 0x1C, 0x24, ... would appear inconsistent to a stride or stream prefetcher that was not designed with complex strides in mind [16]. Searching for timely deltas makes the complexity of the stride pattern irrelevant, if the pattern fits in the IP's history then the timely delta will be a multiple of the sum of the components, without any need to recognize or detect the complex pattern. The previous example of a complex stride alternated the strides 4, 8, 4, 8, ..., which produces timely deltas of 12, 24, ... without any specialized complex stride detection.

Berti collects the timing information it requires to compute timely deltas in two structures (shown in Fig. 2). First, it associates each access in its history table (Fig. 4) with a timestamp, such as a cycle count. The Berti history table is logically an IP-indexed set of FIFO queues holding the access history for each IP. Second, Berti further adds a timestamp to each MSHR, and a latency field to each L1 cache line. When a prefetch is filled, the latency is recorded with the line, or set to zero on an overflow. When the latency information for a demand access becomes available, either because the access hit on a prefetched cache line or because a miss was filled, Berti searches for timely deltas in its history table by looking in the row indexed by the access IP for accesses that are timestamped further in the past than the latency of the access. The differences between the cache line addresses of those past accesses and the line address of the current access are the set of timely deltas. Fig. 3 illustrates this process for a 25-cycle load demanded at timestamp 35, with prior loads at timestamps 15 and 5. When Berti finds timely deltas for an IP, it records them in its delta table.

The delta table (shown in Fig. 5) maps IPs to timely deltas. Every time a delta is found for an IP, its associated `coverage` counter is incremented, as is the global counter for the IP. When the global counter overflows, the `status` of each IP is set according to a series of thresholds. On each memory access, the delta table row for that IP (if any) is searched for deltas with a `PREFETCH` or `PREFETCH_REPLACE` status, indicating deltas that were often timely in the past. Berti uses these high coverage deltas for prefetching by adding them to the access address and issuing prefetches for the resulting cache lines. Deltas with low coverage when the counter overflows are instead marked for replacement by new deltas when they arrive.

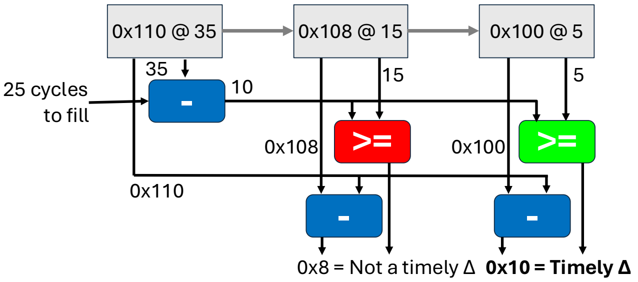

Figure 3: Calculating timely deltas with Berti for one IP with accesses to addresses 0x100, 0x108, then 0x110.

4 GLOBAL BERTI DESIGN

Our Global Berti prefetcher detects *both* streaming and spatial access patterns with small modifications to Berti's history table. It then stores deltas for both types of access patterns in the same delta table with little additional storage, and issues prefetches agnostically to which pattern a delta represents.

4.1 Global Berti Structures

Fig. 4 shows the modifications we make to Berti's history table. The structure of the table is unchanged, we instead add a new way of accessing it. Berti accessed its history table row-wise, inserting new entries at the front of each FIFO row and checking an entire row for timely deltas based on the access history of one IP. We add the option to access the first *column* of the table, corresponding to the most recent accesses by *all* IPs. We perform this column access when we search for global deltas corresponding to spatial access patterns, which we do on each demand access by an IP with no known streaming access patterns.

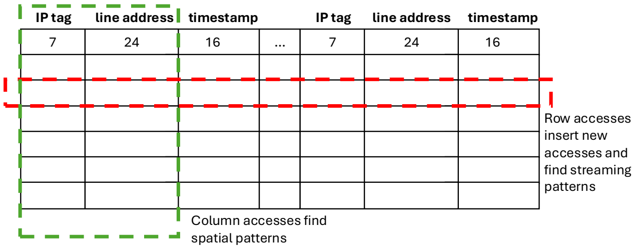
IP tag
line address
timestamp
IP tag
line address
timestamp
7
24
16
...
7
24
16
Row accesses insert new accesses and find streaming patterns
Column accesses find spatial patterns

Figure 4: Global Berti History table with access patterns. Numbers represent field bitwidths.

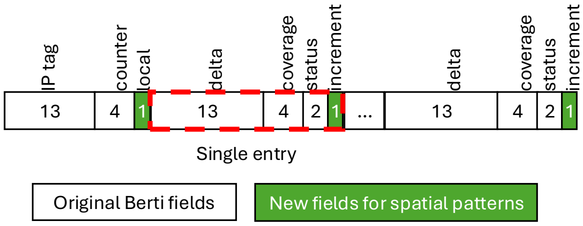

Figure 5: Global Berti Delta table with additions highlighted. Numbers represent field bitwidths.

Fig. 5 shows our delta table, which is also almost identical to Berti's with the addition of a single extra bit per row that tracks the presence of local deltas, and a single bit per delta entry that prevents double-counting. Issuing prefetches using the delta table is similar to Berti. On each demand access, if there are deltas with a PREFETCH or PREFETCH_REPLACE status in that IP's delta table entry, then those deltas are added to the demand line address to determine the target line addresses to prefetch.

4.2 Training Global Berti

Training Global Berti has two stages: *(i) local training* finds streaming patterns with timing information, similarly to the original Berti, and *(ii) global training* finds spatial patterns.

Local training is triggered when access latency information becomes available, either on a cache hit to a line with latency data stored alongside it, or when a demand miss is filled. At this time, like Berti, Global Berti performs a *row access* to its history table, collecting timely deltas. A delta is considered timely if its timestamp is less than the current timestamp minus the latency of the access being considered. These deltas are used to increment the coverage of entries in the delta table for the triggering IP, and the row's counter is also incremented. If the counter overflows, then all deltas in the row have their state set to either NO_PREFETCH, PREFETCH_REPLACE, or PREFETCH based on whether or not they meet the associated thresholds (0, 5, and 8 respectively). The coverages of each entry are then reset.

Local training is where Global Berti sets a delta table row's local field. If any timely delta found for this access has a state other than NO_PREFETCH or a coverage greater than the lowest prefetching threshold, the Global Berti sets the local flag in that delta table row to indicate the presence of a streaming pattern. The local flag disables global training for the delta table row in question. We disable global training when a streaming pattern is present because any IP with a consistent offset from a streaming access is itself also a streaming access, and would therefore receive no benefit from global training while filling its delta table row with useless entries.

Global Berti performs global training on each demand access, without waiting for latency information. Because spatial patterns are detected only for IPs without streaming patterns, Global Berti cannot start prefetching a spatial pattern until the first access in it, which is likely a cache miss. As a result, timing information is not useful for spatial patterns. If Global Berti either does not have an entry in the delta table for the demand IP, or has an entry without the local flag set, it then performs a *column access* in the history table, collecting the *most recent* access in each row. It uses the cache lines of these accesses to calculate deltas, and then attempts to insert those deltas into the delta table entries corresponding to the IPs of those prior accesses, succeeding if the delta table entries both exist and do not have their local flags set.

For both local and global training, Global Berti increments the coverage of a delta at most once per instance of the IP for that row. This is achieved using the increment field that Global Berti adds to each delta entry. The increment fields of all entries in an IP's delta table row are set on each demand access to the IP, at the same time that the row's counter is incremented. When a matching delta is inserted, that delta's coverage is only incremented if the increment field is still set, and the increment field is than cleared. This prevents many accesses by different IPs to the same cache line from immediately saturating the coverage of prior IPs.

Table 1: Storage Requirements of Global Berti

	Entries	Entry size (b)	Total size (B)
History Table	64 × 16	47	6016
Delta Table	64	18 + 32 × 20	5264
MSHR Timestamps	16	16	32
L1 Latencies	768	12	1152
Total			12464

4.3 Global Berti Storage Overheads

Table 1 summarizes the Global Berti storage overheads. We have increased the capacity of both the history and delta tables compared to the original Berti, choosing 64 rows for both our history and delta tables, 16 entries for each history FIFO, and 32 deltas per delta table row. These selections put the total storage budget of Global Berti slightly over 12KB, which still falls well within DPC4's 32KB limit for L1 prefetchers. We leave exhaustive parameter sweeps to future work.

5 EVALUATION

5.1 Methodology

We measure performance using Champsim [10] with the DPC4 single-core full-bandwidth configuration. Table 2 summarizes the system parameters. We measure performance on the entire set of public DPC4 traces, weighted equally. We skip the first 50M instructions of each trace and measure IPC for the next 200M instructions.

We measure speedup as the ratio of IPC against DPC3 Berti [18]. DPC3 Berti differs significantly from Berti [15], as described in

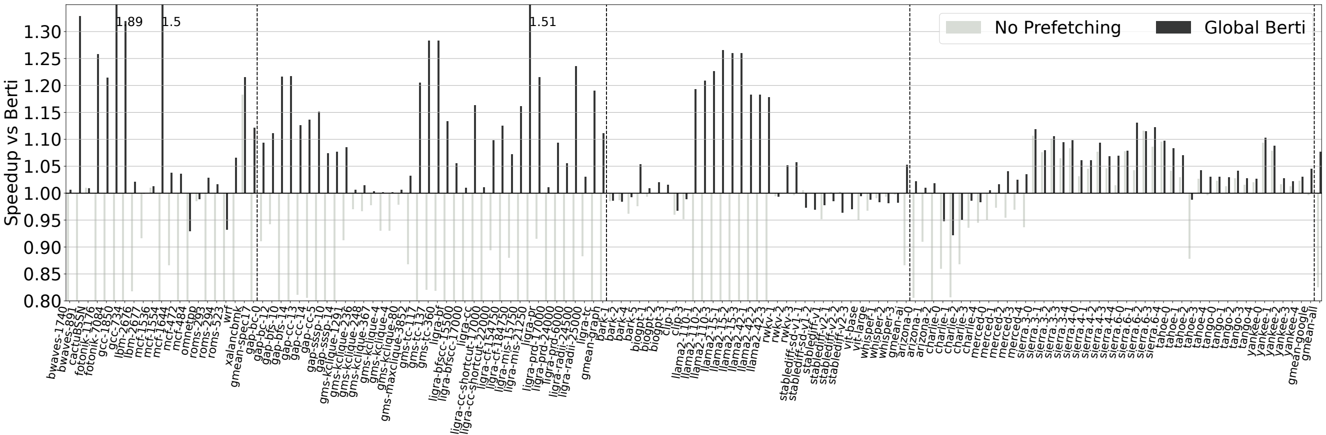

Figure 6: Speedup of L1 Global Berti normalized to L1 Berti with no other prefetchers

Table 2: System Configurations

Core	8-wide OoO, 576-entry ROB, 160 issue slots, 240-entry load queue, 112-entry store queue
L1D cache	48KB, 12-way, 16 MSHRs, 5-cycle latency, Berti/GBerti/No prefetcher
L2 cache	2MB, 16-way, 32 MSHRs, 10-cycle latency, Pythia/No prefetcher
L3 cache	3MB, 12-way, 64 MSHRs, 35-cycle latency, No prefetcher
Main mem	1 channel, 4800 MTPS

Sec. 3. Unlike Berti, DPC3 Berti associates timely deltas with pages instead of IPs, determining the best delta for prefetching within each page. This per-page delta will capture a streaming pattern if one is present, and may partially cover a spatial pattern if there is no streaming pattern touching the page. DPC3 Berti associates IPs with pages, and uses them to generalize deltas to new pages. DPC3 Berti also records the access set of each page, and uses this information to reduce useless prefetches when prefetching within a page by not issuing prefetches to lines not previously demanded. DPC3 Berti also uses this recording to issue a ’burst’ of prefetches when an old page returns to the cache, similarly to Bingo [5].

5.2 Speedup compared to Berti

Fig. 6 shows the speedup of Global Berti over DPC3 Berti across all of DPC4’s public traces with no other prefetchers in the system. Global Berti provides speedups as high as 1.89× over DPC3 Berti, with no slowdowns worse than 0.92×, for a gmean speedup of 1.08×.

The SPEC17 benchmarks observe the largest speedups, with a 1.89× speedup on 602.gcc-1850B, and additional large speedups on mcf, fotonik, and xalancbmk. On the SPEC17 traces, the gmean speedup over baseline Berti is 1.12×. Graph algorithm traces observe almost as large a speedup, with large speedups on ligra Maximal Independent Set and gmc Triangle Counting. Overall, the graph traces observe a gmean speedup over Berti of 1.11×. The AI traces observe no improvement from Global Berti over Berti for most traces, but a large speedup for traces of llama models, resulting in a gmean speedup over Berti of 1.05×. The Google traces, observe a lower but more consistent speedup on all trace categories except charlie, which instead sees a small slowdown compared to Berti. Overall, Global Berti provides a gmean speedup of 1.05× over Berti for Google’s traces.

The largest speedups from Global Berti over Berti come on irregular applications like SPEC17 or graph processing, where streaming access patterns are less important compared to spatial access patterns. AI applications are dominated by matrix multiplies which are highly regular, with no benefit to detecting spatial access patterns over streaming patterns (except on llama). Google’s traces are more heterogeneous [12], so seeing a lower but consistent speedup from prefetching a new type of access pattern makes sense.

5.3 Speedup with Pythia at the L2

Fig. 7 shows the speedups from replacing DPC3 Berti with Global Berti in the DPC4 Baseline configuration, maintaining Pythia at the L2. Pythia [6] is a reinforcement-learning-based prefetcher that learns an offset to prefetch for each miss IP based on memory bandwidth consumption and prefetch usefulness. Because Pythia’s feedback is agnostic as to which IP requests the cache line it prefetched, it can find some of the same spatial patterns as Global Berti. However, as an L2 prefetcher Pythia cannot prefetch outside the same page as the triggering demand access, limiting the offsets it can use. Pythia also issues at most one prefetch per access, which may make it harder for Pythia to be timely for large spatial patterns with many accesses being predictable following one cache miss. Replacing Berti with Global Berti at the L1 while Pythia is active at the L2, we see speedups up to 1.49×, slowdowns no worse than 0.94×, and a gmean 1.06× improvement.

Overall, Global Berti provides lower speedups over Berti with Pythia at the L2 compared to without. This is to be expected from Amdahl’s Law [2] even if there was no overlap between Pythia’s prefetching and Global Berti’s. However, the reduction in speedup from adding Pythia at the L2 varies across trace categories. On the SPEC17 traces, there is little reduction in gmean speedup. The speedup on gcc reduces to 1.49×, but he gmean speedup over Berti+Pythia is still 1.11×. Using Global Berti alone has a large slowdown (gmean 0.93× over Berti+Pythia, indicating that Pythia

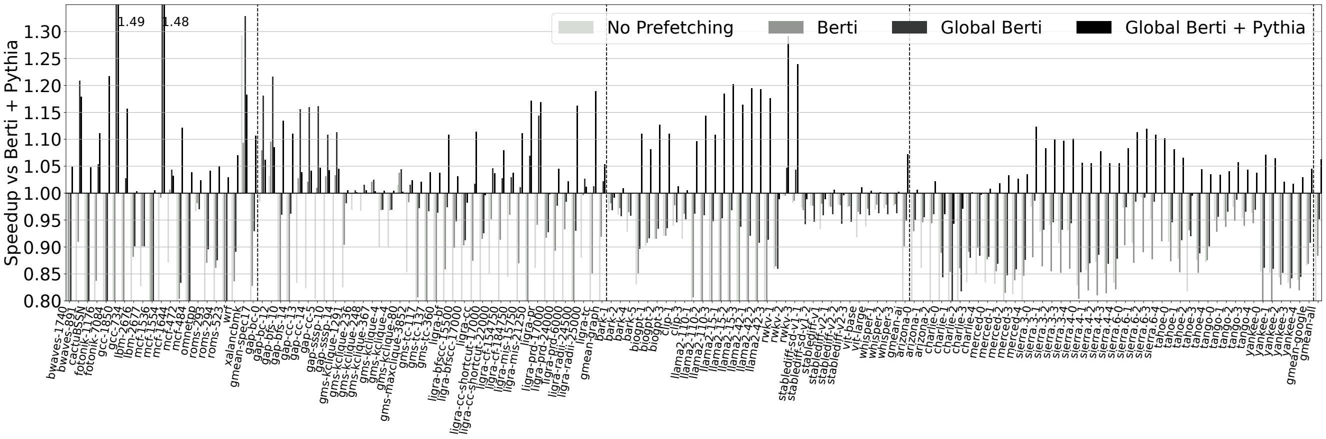

Figure 7: Speedup of L1 Global Berti with and without L2 Pythia normalized to L1 Berti with L2 Pythia

is providing a large speedup over Berti alone on the SPEC17 traces. Pythia's speedup on these traces appears to be mostly orthogonal to Global Berti's, enabling a similar speedup from replacing Berti with Global Berti regardless of whether or not Pythia is contributing.

In contrast, on graph traces, the gmean speedup from Global Berti over Berti reduces to 1.06× when Pythia is at the L2, but using Global Berti without Pythia produces a gmean 1.02× speedup over Berti+Pythia. This indicates that on graph benchmark traces, Global Berti and Pythia are producing similar prefetches, with the benefit of using Global Berti instead of Pythia likely being due to Global Berti being located at the L1 instead of the L2, so its useful prefetches save more cycles. For AI traces, the gmean speedup of Global Berti increases to 1.07×, driven by large speedups appearing for rwkv traces 2 and 3, where Pythia makes no difference for Berti but starts providing a large speedup for Global Berti, suggesting that Global Berti's spatial prefetches are providing useful hints to Pythia's Reinforcement Learning agent. Finally, Google's traces see no significant change in gmean speedup of Global Berti over Berti when Pythia is present at the L2, similarly to SPEC17.

6 RELATED WORK

Streaming prefetchers originate from the basic stride [3] and stream [17, 21]. These prefetchers identify and predict the common programming pattern of striding over an array in a loop, where the same instruction accesses data a consistent offset from the previous access. More sophisticated stride prefetchers such as IPCP [16] can recognize more complex streaming patterns, such as a repeating sequence of strides. Berti [15] introduces the idea of tracking prefetch timeliness to determine how far ahead to prefetch in the stream, while the Micro-Armed Bandit [9] uses a reinforcement learning agent for this purpose.

Spatial prefetchers are most simply exemplified by the next-line prefetcher [19], which simply prefetches a constant delta of +1 for all memory accesses on the basis that programs tend to access memory sequentially. More sophisticated spatial prefetchers such of the Best Offset Prefetcher [14] learn the best global delta for a workload. Spatial prefetchers such as Bingo [5] and Gaze [7] divide the address space into regions (typically pages) and record the cache lines accessed within each region, then replay those accesses if the program returns to the region in question, or appears to be repeating the pattern in a different region. This approach can be effective, but requires high storage overhead and risks issuing many useless prefetches [7]. Less aggressive spatial prefetchers such as Hyperion [8] and DPC3 Berti [18] associate a delta with each page instead of prefetching a full access set. Our approach of associating potentially multiple spatial deltas with each IP is unusual for spatial prefetchers, and allows us to be more accurate than global or regional delta based spatial prefetchers like the Best Offset Prefetcher while avoiding the over-prefetching of record-and-replay prefetchers like Bingo.

7 CONCLUSION

We have presented our submission for DPC4, the Global Berti prefetcher. Global Berti requires almost no additional storage compared to baseline Berti, but is able to detect and predict spatial patterns composed of offsets between IPs as well as streaming patterns between instances of a single IP. By accessing the same history table with different orientations, Global Berti achieves up to 1.89× speedup over baseline Berti for only 264B of additional storage.

ACKNOWLEDGMENTS

This work was supported in part by the Digital Research Alliance of Canada, the University of Toronto Connaught Fund, and the Ontario Graduate Scholarship program.

REFERENCES

[1] Sam Ainsworth and Lev Mukhanov. 2024. Triangel: a high-performance, accurate, timely on-chip temporal prefetcher. In *Proc. of the International Symposium on Computer Architecture* (ISCA-51). ACM/IEEE, 1202–1216. DOI: 10.1109/ISCA59077.2024.00090.

[2] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the American Federation of Information Processing Societies Spring Joint Computer Conference* (AFIPS).

[3] Jean-Loup Baer and Tien-Fu Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of the Conference on Supercomputing* (SC). ACM/IEEE, 176–186. DOI: 10.1145/125826.125932.

[4] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *Proc. of the International Symposium on High Performance Computer Architecture* (HPCA-24). IEEE, 131–142. DOI: 10.1109/HPCA.2018.00021.

[5] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *Proc. of the International Symposium on High Performance Computer Architecture* (HPCA-25). IEEE. DOI: 10.1109/HPCA.2019.00053.

[6] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: a customizable hardware prefetching framework using online reinforcement learning. In *Proc. of the International Symposium on Microarchitecture* (MICRO-54). IEEE/ACM, 1121–1137. DOI: 10.1145/3466752.3480114.

[7] Zixiao Chen, Chentao Wu, Yunfei Gu, Ranhao Jia, Jie Li, and Minyi Guo. 2025. Gaze into the pattern: characterizing spatial patterns with internal temporal correlations for hardware prefetching. In *Proc. of the International Symposium on High Performance Computer Architecture* (HPCA-31). IEEE, 173–187. DOI: 10.1109/HPCA61900.2025.00024.

[8] Yujie Cui, Wei Chen, Xu Cheng, and Jiangfang Yi. 2024. Hyperion: a highly effective page and pc based delta prefetcher. *ACM Trans. Archit. Code Optim.* DOI: 10.1145/3675398.

[9] Gerasimos Gerogiannis and Josep Torrellas. 2023. Micro-armed bandit: lightweight & reusable reinforcement learning for microarchitecture decision-making. In *Proc. of the International Symposium on Microarchitecture* (MICRO-56). IEEE/ACM, 698–713. DOI: 10.1145/3613424.3623780.

[10] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The championship simulator: architectural simulation for education and competition. (2022). https://arxiv.org/abs/2210.14324 arXiv: 2210.14324 [cs.AR].

[11] Shuiyi He, Zicong Wang, Xuan Tang, Hao Tang, Dezun Dong, and Liquan Xiao. 2025. Elevating temporal prefetching through instruction correlation. In *Proc. of the International Symposium on Microarchitecture* (MICRO-58). IEEE/ACM, 915–928. DOI: 10.1145/3725843.3756133.

[12] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proc. of the International Symposium on Computer Architecture* (ISCA-42). ACM/IEEE, 158–169. DOI: 10.1145/2749469.2750392.

[13] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A.L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *Proc. of the International Symposium on Microarchitecture* (MICRO-49). IEEE/ACM, 1–12. DOI: 10.1109/MICRO.2016.7783763.

[14] Pierre Michaud. 2016. Best-offset hardware prefetching. In *Proc. of the International Symposium on High Performance Computer Architecture* (HPCA-22). IEEE, 469–480. DOI: 10.1109/HPCA.2016.7446087.

[15] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an accurate local-delta data prefetcher. In *Proc. of the International Symposium on Microarchitecture* (MICRO-55). IEEE/ACM, 975–991. DOI: 10.1109/MICRO56248.2022.00072.

[16] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: instruction pointer classifier-based spatial hardware prefetching. In *Proc. of the International Symposium on Computer Architecture* (ISCA-47). ACM/IEEE. DOI: 10.1109/ISCA45697.2020.00021.

[17] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. 2020. Reverse engineering the stream prefetcher for profit. In *Proc. European Symposium on Security and Privacy Workshops* (EuroS&PW). IEEE. DOI: 10.1109/EuroSPW51379.2020.00098.

[18] Alberto Ros. 2019. Berti: a per-page best-request-time delta prefetcher. *The 3rd Data Prefetching Championship*.

[19] A.J. Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer*. DOI: 10.1109/C-M.1978.218016.

[20] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52, 4, 65–76. DOI: 10.1145/1498765.1498785.

[21] Chengqiang Zhang and Sally A. McKee. 2000. Hardware-only stream prefetching and dynamic access ordering. In *Proc. of the International Conference on Supercomputing* (ICS). ACM, 167–175. DOI: 10.1145/335231.335247.