The Entangling Data Prefetcher

Agustín Navarro-Torres*
agusnt@unizar.es
Universidad de Zaragoza
Zaragoza, Aragón, Spain

Simranjit Singh†
simranjit.singh@um.es
University of Murcia
Murcia, R. Murcia, Spain

Biswabandan Panda‡
biswa@cse.iitb.ac.in
Indian Institute of Technology Bombay
Bombay, Maharashtra, India

Alberto Ros†
aros@ditec.um.es
University of Murcia
Murcia, R. Murcia, Spain

Abstract

The performance of a prefetcher is determined by its coverage and timeliness. If a correct address is prefetched too late, the potential to hide latency diminishes. Recent data prefetching techniques address timeliness by adjusting the stride or delta used to trigger prefetch requests, or by decoupling the instruction that triggers the requests ($IP_{trigger}$) from the one used to predict the pattern (IP_{target}). We observe that state-of-the-art prefetchers that decouple $IP_{trigger}$ and IP_{target} often fail to achieve timeliness or cover complex patterns; meanwhile, prefetchers that use the same $IP_{trigger}$ and IP_{target} struggle with zero-strides and long-reuse patterns.

This work proposes the Entangling Data Prefetcher (EDP), a novel L1D prefetching technique that combines the strengths of delta-based prefetchers by decoupling $IP_{trigger}$ and IP_{target}. In general, EDP leverages local deltas to cover complex delta patterns, and when the trigger and target instructions differ, it can successfully cover long-reuse patterns and zero-delta patterns. On average across the full bandwidth configuration and the 4-core configuration, EDP outperforms a baseline with Berti at L1D and Pythia at L2 by 7.1% and 1.2%, respectively.

1 Introduction and Motivation

State-of-the-art prefetchers learn memory access patterns and issue prefetch requests using the same instruction pointer (IP) [1, 3, 6–8]. In this paper, we focus on two of these prefetchers: Berti and T-SKID, which attempt to address timeliness in distinct ways.

Berti [6] is an L1D prefetcher that provides high prefetch accuracy and coverage by using a mechanism of prefetching local (per-IP) deltas. In this context, a local delta is defined as the difference in cache line addresses between two L1D demand accesses issued by the same instruction (IP). To convert an L1D miss into an L1D hit, Berti estimates the fetch latency for data from outer cache levels and DRAM. Berti selects only the deltas that can bring data into the L1D in a timely manner. Alongside these timely local deltas, it computes the local coverage provided by each delta and issues prefetch requests that use the deltas offering maximum coverage. Based on this same metric, it also orchestrates whether prefetch requests fill the L1D or the L2 cache.

T-SKID [4] is built on top of an IP-stride prefetcher and focuses on delaying early prefetch requests. Its key concept is decoupling the IP that triggers a prefetch request ($IP_{trigger}$) from the IP that generates the access and trains the predictor (IP_{target}). This decoupling allows T-SKID to cover two patterns previously unaddressed: (1) patterns with long reuse distances that stride or delta prefetchers can predict and issue, but which are evicted before a demand access occurs; and (2) zero-stride [1] patterns, where two consecutive misses from the same IP access the same cache line.

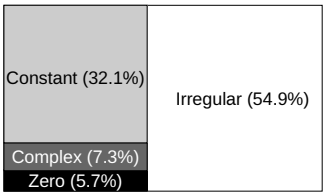


Figure 1: Characterization of patterns for SPEC CPU2017.

When Berti and T-SKID fail to prefetch. Figure 1 shows a classification of per-IP stride patterns for L1D misses and their frequency in SPEC CPU2017 benchmarks: a *zero* stride; a *constant* stride, that occurs when the current miss has the same stride as previous misses; a *complex* stride, that refers to multiple repeating, predictable strides; and *irregular*, that covers all other remaining misses. Berti is unable to capture long-reuse and zero-delta patterns (5.7%) because the $IP_{trigger}$ and the IP_{target} are the same, while T-SKID fails to learn complex strides (7.3%) because it is built on top of an IP-stride prefetcher [5–7].

This limitation is exemplified in trace `607.cactuBSSN-2421B` by IP `0x8d41a1` (Figure 2). This IP exhibits a predictable but complex stride pattern (+0, +0, +0, +1, +0, +0, +0, +1, …). Accesses are separated by a 2K-cycle interval, and cache lines are evicted between accesses, resulting in a 100% miss rate (even for the +0 stride) at L1D. Berti can correctly identify this kind of pattern, but it suffers from the inability to issue zero-delta requests. In addition, although Berti can issue prefetch requests using a +1 delta, the lines would arrive too early and would be evicted before their demand access, resulting in a 100% miss rate. T-SKID can issue timely prefetch requests correctly for this access pattern. However, it cannot learn this complex stride pattern. It may capture the +0 stride or the +1 stride, but not both simultaneously. As a result, T-SKID can only cover up to 75% of the accesses (when learning +0).

2 The Entangling Data Prefetcher

The Entangling Data Prefetcher (EDP) is a data prefetcher situated at the L1D. It monitors all CPU-generated data memory requests, analyzes the IP's access patterns, identifies the most suitable $IP_{trigger}$

[1] We use the term 'zero-stride' for stride-based prefetchers (e.g., IP-stride) and later 'zero-delta' for delta-based prefetchers (e.g., Berti).

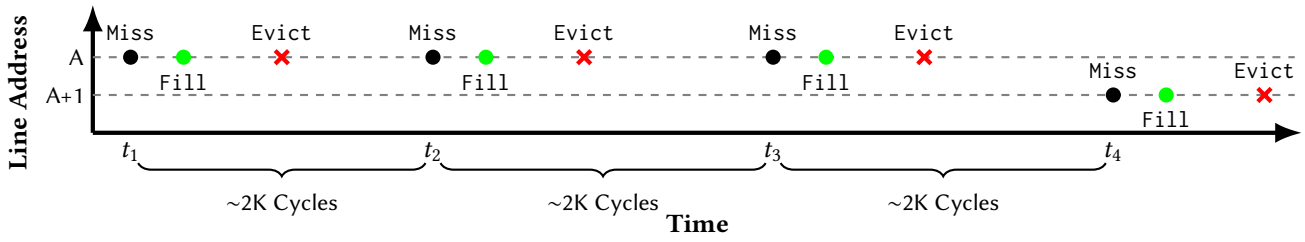


Figure 2: Temporal diagram of memory access patterns for IP `0x8d41a1` from trace `607.cactuBSSN-2421B`. Recurrent cache misses are observed on Line A with a delta of +0, triggering eviction events on the adjacent line (Line A+1) at intervals of approximately 2K cycles.

for issuing prefetch requests, and orchestrates prefetch operations across the cache hierarchy. EDP makes a strong case for determining *when* prefetch requests should be issued. For each IP, EDP selects a prior IP that can trigger a prefetch request on its behalf. Additionally, EDP learns timely local deltas to decide *what* to prefetch.

EDP in a nutshell. EDP combines the pattern detection capabilities of delta prefetching with precise timing control through instruction entangling, and can fully cover memory access patterns that were uncovered with state-of-the-art prefetchers like Berti. EDP is highly inspired by Berti and T-SKID. It is based on the observation that combining the decoupling of $IP_{trigger}$ and IP_{target} with the use of local deltas for learning memory patterns allows EDP to cover zero-deltas and long-reuse patterns by delaying prefetch requests, while simultaneously covering complex patterns via delta analysis. EDP introduces a new structure called Proxy Prefetch Queues (PPQ), which enhanced the original PQ. The PPQ consists of two independent queues for L1D and L2 prefetch requests and holds the requests until L1D cache resources—such as MSHRs—become available. In addition, a Bloom filter is used to drop prefetch requests for addresses that have been recently accessed or prefetched. This reduces the potential harmful side effects of added contention. In particular, EDP builds on the following concepts: (i) for each IP, it learns the best local delta to prefetch; (ii) it then searches for earlier IPs that can trigger those deltas; and (iii) it carefully selects when to issue a prefetch request to minimize contention within the memory hierarchy.

2.1 Gathering information

The training mechanism aims to identify the *when* and the *what*, that is, the prior IP capable of issuing a timely prefetch request (denoted $IP_{trigger}$) for the current cache miss or a hit resulting from a prefetch request (prefetch hit), and the timely local deltas to prefetch from the IP that misses the cache (denoted IP_{target}). The training process begins with a cache miss and ends with a demand cache fill or a prefetch hit, and consists of five steps: (i) learning the fetch latency, (ii) learning the $IP_{trigger}$, (iii) learning the local-deltas, (iv) correlating $IP_{trigger}$ and IP_{target}, and (v) updating the history.

2.1.1 Learning the fetch latency. EDP calculates fetch latency in a similar manner to Berti [6]. For every demand miss or prefetch request, it stores the timestamp of the event in the MSHR. When the data returns to the L1D cache, the controller subtracts the stored timestamp from the current cycle. The resulting value represents the fetch latency.

2.1.2 Learning the $IP_{trigger}$. EDP needs to identify the prior IPs that can trigger prefetch requests with sufficient time to ensure that the data fill the cache before being demanded by IP_{target}. To this end, EDP maintains a global history (denoted *Instruction history*) that tracks recent IPs that had a cache miss or a prefetch hit, along with their access timestamps ($IP_{timestamp}$). At fill time, similar to EIP [9], EDP searches the Instruction history for entries whose timestamps are older than the current cycle minus the fetch latency ($IP_{timestamp} \leq current\_cycle - fetch\_latency$). Only the most recent IPs that meets the timeliness criteria are selected as $IP_{trigger}$.

2.1.3 Learning the Deltas. Once the $IP_{trigger}$ is found, EDP learns timely and accurate local deltas of the IP_{target}. EDP maintains a local history indexed by the IP (denoted *Address history*) that tracks the line addresses of previous cache misses and prefetch hits, along with their timestamps ($Addr_{timestamp}$). After the previous step, EDP searches the Address history for IP_{target} and selects entries whose timestamps are less than the current time minus the fetch latency ($Addr_{timestamp} \leq current\_cycle - fetch\_latency$). The deltas are obtained by computing the difference between the current address and the stored address in the history (both addresses are accessed by IP_{target}). All computed deltas, including the delta zero when $IP_{trigger}$ differs from IP_{target}, are saved in a table indexed by the IP (denoted *Delta table*). Similarly to Berti, the coverage of deltas is calculated as the number of times a delta is observed divided by the total number of times the IP searches for deltas. We establish three coverage thresholds for deltas: (i) high coverage, when the delta appears in most observations; (ii) medium coverage, when the delta appears intermittently; and (iii) low coverage, when the delta rarely appears.

2.1.4 Correlation between $IP_{trigger}$ and IP_{target}. When EDP identifies an $IP_{trigger}$, it records the $IP_{trigger}$–IP_{target} relationship in the *Entangling Table*, which consists of two fields: a tag based on the $IP_{trigger}$ and a set of IP_{target}. When a new $IP_{trigger}$ is identified, the Entangling table is indexed using a hash of $IP_{trigger}$, and the associated IP_{target} is inserted if it was not already present in the set. The set of IPs enables a many-to-many relationship between $IP_{trigger}$ and IP_{target}, which allows effectiveness even if some entangled pairs lack correlation, as other pairs will trigger prefetch requests, thus maintaining high coverage.

2.1.5 Updating the histories and the prefetch filter. The final step of the training process involves updating the Instruction History by recording the current IP and timestamp, as well as the Data History with the current IP, line address, and timestamp. EDP maintains a Bloom filter structure called a prefetch filter, which is used to drop prefetch requests for addresses that have been recently accessed or prefetched. The prefetch filter is accessed using the line address to set the corresponding bit in an entry. An entry is cleared only when the cache line associated with it is evicted from the L1D cache.

2.2 Issuing prefetch requests

EDP functions as a virtual L1D prefetcher capable of issuing prefetch requests on every cache access. A primary focus of EDP is to remain outside the demand request path; to this end, it utilizes Proxy Prefetch Queues (PPQ) and a prefetch filter to determine when and whether to issue a request, thereby minimizing contention during high-traffic periods. To issue a prefetch request, EDP performs the following steps: (i) it checks if the $IP_{trigger}$ has any associated IP_{target}; (ii) it retrieves the deltas and the last address of the IP_{target} to generate the prefetch request addresses; (iii) it inserts the generated requests into the PPQ; and (iv) when a read port becomes available in the L1D cache, EDP checks the MSHR occupancy and the prefetch filter to decide whether issue the request.

2.2.1 Recovering the IP_{target} and the Deltas. When a cache miss or a prefetch hit occurs, EDP first checks whether the accessing IP (EDP does not distinguish between multiple instances of an IP) has any associated IP_{target}. If no IP_{target} is associated, no further action is taken. Otherwise, for each associated IP_{target}, EDP retrieves the IP_{target}'s last seen address (from *the Last Address Table*) and its timely deltas (from the *delta Table*). For each delta, EDP selects the cache level (L1D or L2) to fill the prefetch request according to the delta coverage (high/medium). High-coverage deltas are temporally marked to be prefetched into L1D, while medium-coverage deltas target L2 fills. Once the prefetch requests are generated, they are inserted in the to-L1D PPQ or to-L2 PPQ based on their confidence. When the PPQ is full, the oldest entry is removed to accommodate the new one. After generating prefetch requests, the $IP_{trigger}$-IP_{target} association is discarded to prevent the issue of duplicate prefetch requests.

2.2.2 Proxy PQ and filtering redundant requests. On every cycle, EDP monitors the occupancy of the Read Queue (RQ) and the Prefetch Queue (PQ). If the number of entries in either queue exceeds the number of available cache read ports, no prefetch requests are issued. Otherwise, the 'to-L1D' PPQ is accessed, and a number of entries equal to the available read ports are popped and inserted into the cache PQ if the prefetch address was not recently seen by the cache (the prefetch filter associated to the prefetch address is equal to 0). Prefetch requests are tagged to fill the L1D only if the MSHR occupancy is below a specific threshold at the moment of insertion. If the 'to-L1D' PPQ is empty, this same process is performed for the 'to-L2' PPQ. When a prefetch request is generated, the prefetch filter is accessed and its entry set to one.

2.3 L2 and LLC prefetchers

EDP uses Pythia [2] as the L2 prefetcher and a small prefetch mechanism in the LLC to be aware of the general state of the system. This LLC mechanism monitors the number of accesses per core and the ratio of demand and prefetch misses compared to the total fills of the LLC. A low ratio indicates that the core is struggling with other request types (such as PTW), and the L1D prefetcher has to be throttled.

3 Championship Constraints

In this section, we show how our prefetcher complies with the limitations of the 4th Data Prefetching Championship. First, we explain in detail the different structures and their storage overhead, and then we describe how we adjust to the championship guidelines.

3.1 Storage overhead

EDP is sited at L1D and has a total storage overhead of 31.98KB. Table 1 discloses the storage requirements for each component. EDP is composed of eight structures. Figure 3 illustrates the various structures and their corresponding access methods. Next, we describe all the structures:

Cache, PQ, and MSHR: Similar to Berti, EDP extends the cache, MSHR, and PQ with a 16-bit timestamp to calculate the fetch latency. For the DPC configuration, this results in an overhead of 1.55 KB (across 768 cache entries, 16 MSHR entries, and 8 PQ entries).

Instruction History: The instruction history records the previous IPs that were missed in the cache. It is a 32-entry fully associative FIFO queue; each entry stores a 14-bit IP tag and a 16-bit timestamp.

Data History: The data history saves the previous addresses accessed by an IP in order to learn delta patterns. It is a 1024-entry table organized into 32 sets. Each set consists of a 32-entry FIFO queue indexed by the IP. Each entry in the FIFO queue stores a 14-bit IP tag, a 16-bit timestamp, and a 12-bit address.

Delta Table: A 512-entry fully associative structure that saves the learned deltas for each IP_{target}. It is indexed by the IP, and every entry has a 14-bit tag, a 4-bit global confidence, and capacity to store 16 deltas. For every delta, EDP saves the delta value (7 bits plus 1 sign bit), a 4-bit confidence, a valid bit, and 2 bits indicating if the delta will fill L1D or L2, is replaceable, or is a new delta.

Last Addr. Table: A 32-entry fully associative structure that stores the last address seen by each IP to generate the correct prefetch address. Every entry has a 14-bit IP tag, 58 bits for the last seen address, 1 bit to indicate if the IP access delta is positive (ascending) or negative (descending), and a pointer to the Delta Table entry associated with the IP tag. We allow aliasing in this pointer, so no valid bit is necessary.

Entangling Table: A 64-entry fully associative structure to correlate the $IP_{trigger}$ and IP_{target}. It is indexed by the IP, and every entry contains a 14-bit IP tag and four pointers to the Last Addr. Table (5 bits each); each pointer has a valid bit.

Proxy Prefetch Queue: The virtual prefetch queue consists of two 16-entry FIFO queues. Each entry contains only the address of the prefetch address line (58 bits); an entry of all zeros represents a non-valid entry.

Prefetch Filter: The prefetch filter contains 8K single-bit entries.

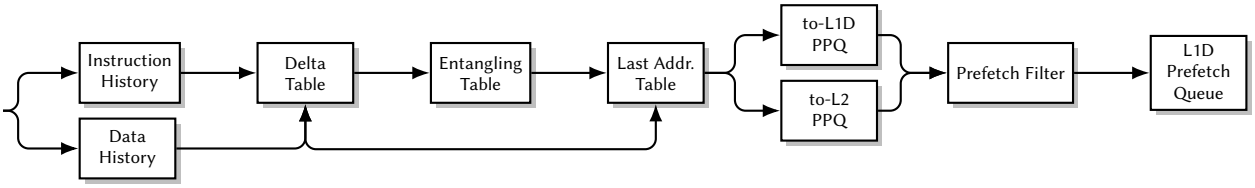


Figure 3: Relation between the different structures of EDP

Structure	Description	KB
Cache, PQ, and MSHR	16 bits per entry (768 L1D entries, 8 PQ entries, and 16 MSHR entries).	1.55
Instruction History	32 entries plus 5 replacement bits. Each entry: 14-bit IP and 16-bit timestamp.	0.12
Data History	1024 entries (32 sets × 32 ways) plus 160 replacement bits. Each entry: 14-bit IP, 16-bit timestamp, and 12-bit address.	5.27
Delta Table	512 entries. Each entry: 14-bit IP, 4-bit confidence, 1-bit for replacement, and 16 delta slots (each containing: 8-bit delta, 4-bit confidence, 2-bit level, 1 valid bit)	16.19
Last Addr. Table	32 entries. Each entry: 14-bit IP, 1 direction bit, 58-bit last seen address, 1-bit for replacement, and 9-bit pointer to Delta Table.	0.32
Entangling Table	64 entries. Each entry: 14-bit IP, 1-bit for replacement, plus 4 pointers (each comprising a 5-bit index to Last Addr. Table and 1 valid bit).	0.30
Prefetch Filter	8192 8-bit entries.	8.00
Proxy Prefetch Queue	2×16 entries plus 4-bit replacement bits. Each entry is 58 bits.	0.23
Other Counters	A 4-bit counter for the prefetch degree.	< 0.01
L1D Total		31.98
L2	Pythia, as given in the Champsionship	25.5
LLC	4 × 4 × 5-bit counters, 2-bit throttle counter, 2-bit multicore, and 256-entry MSHR (58 bits/entry).	1.83

Table 1: Storage requirements of EDP

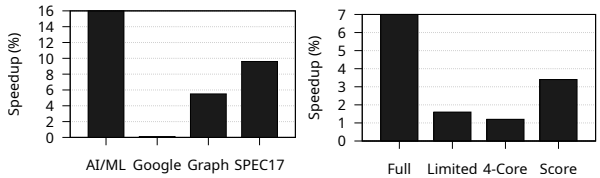
3.1.1 L2 and LLC prefetchers. For the L2 cache, we use the provided version of Pythia. For the LLC mechanism, we require a 256-entry fully associative table (58 bits per entry) to track demand and prefetch requests that miss the cache until they are filled. Additionally, four 5-bit counters and a 2-bit counter monitor per-core accesses to identify concurrent LLC usage by all cores and to set L1D prefetcher aggressiveness. An extra 2-bit counter is used to determine if the L1D prefetcher should be throttled to a maximum degree of 16 or 4, or turned off.

3.2 Use of internal cache structures

Our prefetcher uses the provided cache interface to monitor the occupancy and size of the L1D PQ, RQ, and MSHR. Additionally, we mimic the L1D cache, PQ, and MSHR as required by Berti to calculate fetch latency without modifying the ChampSim code. At no point do we use these mimics to check if a cache line is already present in the cache; this is achieved through the prefetch filter.

4 Evaluation

We evaluate our prefetcher (EDP) using the three configurations provided for the Championship: full bandwidth, limited bandwidth, and 4-core. We use the traces provided by the competition, grouped in AI/ML, Google traces (Google), Graph, and SPEC17. Fig. 4a shows the per-group average speedup for the full bandwidth configuration. Fig. 4b shows the average speedup across the three different configurations and the overall score following the Championship metrics. For the 4-core simulations, we use 50 randomly created heterogeneous mixes from all available traces.



(a) Full bandwidth **(b) Overwiew**

Figure 4: Speedup with EDP w.r.t DPC-4 baseline

5 Discussion and Future Work

In the submitted version of the prefetcher, we employ mostly fully associative structures, increasing the size of the structures as much as possible. However, this is not a feasible implementation.

Our implementation only explores the occupancy of PQ, RQ, and MSHR to issue prefetch requests within the context of the LLC. However, virtual prefetchers add pressure to the TLB; adding a throttling mechanism to take into account the state of the TLB is left for future work to continue pushing performance.

6 Acknowledgments

This work was supported by MICIU/AEI/10.13039/501100011033 (grants PID2022-136454NB-C22 and PID2022-136315OB-I00), by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 819134), and by the Government of Aragon (T58_23R research group).

References

[1] Jean-Loup Baer. 2009. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors* (1st ed.). Cambridge University Press.

[2] 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 *54th Int'l Symp. on Microarchitecture (MICRO)*. 1121–1137.

[3] 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 *31th Int'l Symp. on High-Performance Computer Architecture (HPCA)*.

[4] Toru Koizumi, Tomoki Nakamura, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. 2022. T-SKID: Predicting When to Prefetch Separately from Address Prediction. In *25thDesign, Automation, and Test in Europe (DATE)*. 1389–1394.

[5] Pierre Michaud. 2016. Best-offset hardware prefetching. In *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 469–480.

[6] 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 *55th Int'l Symp. on Microarchitecture (MICRO)*. 975–991.

[7] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñnals-Yúfera, and Alberto Ros. 2025. A Complexity-Effective Local Delta Prefetcher. *IEEE Transactions on Computers (TC)* (2025), 1–12.

[8] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *47th Int'l Symp. on Computer Architecture (ISCA)*. 118–131.

[9] Alberto Ros and Alexandra Jimborean. 2021. A Cost-Effective Entangling Prefetcher for Instructions. In *48th Int'l Symp. on Computer Architecture (ISCA)*. 99–111.