Pushing the Limits of the Berti Prefetcher

Abstract

Data prefetching techniques are commonly employed at several levels in the cache hierarchy, due to their importance in reducing the latency of memory operations. Berti is one of the state-of-the-art prefetching mechanisms, originally proposed for the 3rd Data Prefetching Championship (DPC-3) for all cache levels. A later revision uploaded to DPC-3 limited Berti to the first and second cache levels. An improved version of Berti using virtual addresses was proposed only at the first level data cache, but orchestrating prefetchers across the cache hierarchy.

The Berti prefetcher is one of the state-of-the-art prefetching techniques, and despite doing accurate predictions, it wastes resources in the cache hierarchy and does not leverage context information. For example, Berti issues a large number of replicated prefetch requests, many times for lines that are already in cache. In addition, it only uses the current instruction pointer (IP) as context information, missing patterns that could be recognized if the context were considered.

This work pushes Berti to its limits with two enhancements: a Region-based bit-map filter that eliminates both redundant and useless prefetch requests, and the use of IP-path signatures to capture context information. This prefetcher is used in the first-level data cache. We complement our enhanced Berti prefetcher with a Pythia prefetcher improved with set dueling at the second-level cache, which dynamically selects among four feature policies plus a no-prefetch option. Similarly, in the last-level cache, we model a dynamic next-line prefetching mechanism. For the set of training traces of the 4th Data Prefetching Championship, we achieve 10.4% average performance improvements over the given baseline on full bandwidth, reaching 30.3%, on average, for AI/ML workloads. For the limit bandwidth configuration, average improvements of 4.5% are obtained.

1 Introduction and Motivation

Modern data prefetchers have matured significantly, pushed by recent Data Prefetching Championships, yet even state-of-the-art designs have room for improvement. The 4th Data Prefetching Championship (DPC-4) provides a baseline prefetcher where it pairs a Berti prefetcher published at the 3rd Data Prefetching Championship (DPC-3) [7] located in the first-level data cache (L1D) with a Pythia prefetcher [1] located in the second-level cache (L2).

The DPC-3 Berti prefetcher was trained on physical addresses and required a non-negligible amount of logic and storage to connect physical pages as accessed by the programs. A subsequent version of Berti [4] simplified the design, operating with virtual addresses. Furthermore, important performance improvements were obtained by detecting local timely deltas (per instruction pointer —IP—, instead of memory regions) and by adding a mechanism to select prefetch requests based on their expected accuracy. Our first step is therefore to port the latest version of Berti [4] to the DPC-4 infrastructure.

Then, we identify two inefficiencies of the Berti prefetcher, that we address in this work: the redundancy tax and the lack of context. The redundancy tax arises because Berti issues prefetches for lines already in cache, wasting prefetch queue slots and cache port bandwidth. The lack of context stems from Berti indexing by IP alone, missing patterns that depend on how execution reached that instruction.

We address the redundancy tax with a Region-based prefetch filter in the first-level data cache (L1D) prefetcher. The filter tracks recently accessed or prefetched cache lines within each memory page. For prefetch requests targeting the L1D, the filter eliminates redundant prefetches, that is, requests for lines already in cache or in flight. Without filtering, these requests waste prefetch queue slots and cache port bandwidth. For prefetch requests targeting the second-level cache (L2), the filter additionally learns to suppress useless prefetches, that is, requests for lines that are never used. If a prefetched line is evicted from L2 without being promoted to L1D, its bit remains set, blocking future prefetches to the same address until the filter entry is evicted.

To capture execution context, we augment the instruction pointer with a IP-path signature. By folding the current IP together with the last four IPs into a single hash, we create an additional lookup key that distinguishes the same instruction reached via different execution paths. This enables Berti to learn separate patterns for each calling context without significantly increasing storage.

The static policies at L2 and LLC also leave performance on the table. At the L2, we observe that Pythia's default feature policy does not suit all workloads. We introduce set dueling, originally proposed by Qureshi et all [6] but adapted to prefetching, to let the workload itself select among four feature policies plus a no-prefetch option during a short tournament phase. At the last-level cache (LLC), we deploy ANeLin, an adaptive next-line prefetcher that learns both globally and per-IP whether next-line prefetching is beneficial.

On the DPC-4 training traces, our complete stack achieves 10.4% average speedup under full bandwidth and 4.5% under limited bandwidth. AI/ML workloads benefit the most, reaching 30.3% speedup on full bandwidth.

2 The L1D Prefetcher: BertiGO

BertiGO addresses the redundancy tax and context limitations with two mechanisms, a Region-based prefetch filter that suppresses redundant and useless prefetches, and IP-path signatures that capture execution context beyond the current IP.

2.1 Region-Based Bit-Map Filter

We introduce a Region-Based Bit-Map Filter, organized as a cache-like structure with multiple entries. Each entry contains a tag identifying a 4KB page and a 64-bit bitmap covering the 64 cache lines within that page. When a line is accessed or prefetched, we set the corresponding bit in the matching entry. Before issuing a prefetch, Berti looks up the filter by page tag. If an entry exists and the target line's bit is set, the request is suppressed. Entries are managed with

NRU replacement and cleared on L1 evictions. Our filter uses 15.6 KB (1360 entries, each with a 29-bit tag and 64-bit bitmap, tracking up to 87,040 cache lines). Smaller configurations capture most of the benefit at a fraction of the storage. The filter serves two purposes. First, it eliminates redundant prefetches, and second, it learns to suppress useless ones.

A prefetch is redundant when the target line is already in cache or in flight. Without filtering, these requests waste prefetch queue slots and cache port bandwidth. Note that redundant requests are not necessarily useless, as they serve to hint the replacement policy that the line will be reused soon [8]. However, in an aggressive prefetcher like Berti, they compete for resources with prefetches for lines not yet in cache.

The filter also learns to suppress useless prefetch requests. L2 prefetches have lower confidence, so some bring lines that are never used. If such a prefetch is never promoted to L1, no L1 eviction clears its bit, and future prefetch requests to that line are blocked until the entry is evicted by capacity. In contrast, L1 prefetches require higher confidence and may simply be early. We clear their bits on L1 eviction to allow retries.

A desirable property of this structure is that it eliminates aliasing at low cost. A region tag of sufficient width ensures no false positives within the tracked address space. Our filter uses 15.6 KB (1360 entries, each with a 29-bit tag and 64-bit bitmap, tracking up to 87,040 cache lines). Alternative designs such as Bloom filters would require substantially more storage for equivalent coverage if we desire low aliasing. For instance, using the Bloom filter formula [3] $n = \lceil m/(-k/\ln(1 - e^{\ln p/k})) \rceil$, where n is the number of tracked items, m is the filter size in bits, k is the number of hash functions, and p is the false positive probability, a filter tracking 87,040 cache lines with $p = 0.01$ and $k = 3$ requires approximately 131 KB, nearly 8× our 15.6 KB, and still inherently has aliasing. Furthermore, the structure can be organized with different set/way configurations; smaller filters capture most of the benefit and remain hardware-friendly. For instance, a direct-mapped version would allow adding entries or filtering all entries within a region at once before adding to the prefetch queue with simple bitwise operations per page.

2.2 IP-path Signatures

The same instruction can exhibit different memory access patterns depending on how execution reached it. For example, a load instruction in a shared utility function may access different data structures depending on its caller. Indexing by IP alone mixes these distinct patterns, limiting predictability.

To capture this context, we augment the instruction pointer with a IP-path signature. By folding the current IP together with the last four IPs using shift-and-XOR, we create a hash that distinguishes the same instruction reached via different execution paths. This enables Berti to learn separate delta patterns for each calling context. At lookup, we query the history and delta tables with both the IP alone and the IP-path signature. Predictions are merged, prioritizing IP matches over IP-path matches, as we have seen IP-path being most useful when IP alone lacks sufficient context. More sophisticated merging strategies can be explored in future work. Storage overhead is 8 bytes (4 IPs × 16 bits each).

We observe that this context is particularly effective for AI/ML workloads. On Llama inference traces under full bandwidth, IP-path signatures improve speedup by 20-50% compared to IP-only indexing.

3 The L2 Prefetcher: Pythia with Set-Dueling

Pythia's default policy issues a prefetch whenever either IP or IP_Delta votes yes. We observe that this hurts some workloads. To address this, we apply set-dueling. Originally, set-dueling has been proposed for cache replacement policies and samples competing policies on dedicated sets to selects the best. As far as we know, this approach has not been applied to prefetcher techniques. The 2048 L2 sets are distributed across five candidates: NoPrefetch, IP-only, IP_Delta-only, IP or IP_Delta, and IP and IP_Delta. During a 10M-instruction tournament, each set operates under its assigned policy while we track miss rate (misses divided by accesses) per candidate. At tournament end, we select the candidate with the lowest miss rate, provided it improves over NoPrefetch by at least 4%; otherwise we disable Pythia entirely. All sets then adopt the winning policy for the remainder of execution.

A production implementation could periodically re-run the tournament or trigger it on phase changes; we found DPC-4 traces sufficiently stable to commit once. Per practical considerations, we did not explore fine-tuning to exploit other feature combinations (Page, Delta_Path, etc.). Further combinations can be explored. This mechanism adds negligible storage (ten 64-bit counters) and eliminates L2 prefetch traffic for workloads where it hurts more than it helps.

4 The LLC Prefetcher: Adaptive Next-Line

We observed that some traces benefit from next-line prefetching at the LLC, while it is detrimental for performance for other traces. Therefore, we propose the use of an Adaptive Next-Line prefetching mechanism (ANeLin) at the LLC.

ANeLin uses a sampling cache where next-line prefetch requests are always inserted, similar in spirit to the Sandbox prefetching[5]. The size and associativity of the sampling cache can be smaller than the real cache, a necessary property given the large size of an LLC. On each entry, the cache stores the status (prefetched, demanded, timely, late), the IP that allocated the line in the cache, and the core that allocated the data in the cache (only used in the multicore configuration).

The status is set to *demand* for demanded cache misses, and to *prefetched* for next-line prefetch requests that miss in the sampling cache. On a hit in the sampling cache from a request from L2, if the status is *prefetched*, it changes to *timely* or *late* depending on the time since the prefetch request was issued.

On an eviction from the sampling cache, the status is checked, and a set of three counters are increased. *Timely* entries increase a counter of usefulness. *Late* entries increase the same counter with probability 50%. *Prefetched* entries increase a counter of useless prefetch requests. Any eviction increases a counter of evictions.

These counters are used both globally and per IP in order to decide if the application has a global favoring pattern or if some IP are favoring or not next-line prefetching. The global counter is independent per core in the multicore configuration. IPs are also

Structure	Description	KB
Cache, PQ and MSHR	16 bits per entry.	1.55
History Table	48 sets × 32 ways (1536 entries), plus 5 bits FIFO replacement per set. Each entry: 16 bits IP tag, 24 bits address, 16 bits timestamp.	10.52
Delta Table	64 entries plus 6 replacement bits. Each entry: 16 bits IP, 4 bits confidence, and 16 delta entries (each: 8 bits delta, 4 bits confidence, 2 bits level, 1 bit valid).	2.03
Region-based prefetch filter	1360 entries (fully associative), plus 1360 bits NRU. Each entry: 29 bits tag, 64 bits bitmap.	15.6
Last 4 IPs		<0.01
L1D Total		**29.72**
Pythia		25.5
Set-Dueling Counters	5 candidates × 2 counters (64 bits each), 3 bits winner, 1 bit phase, 64 bits instruction counter.	0.09
L2 Total		**25.30**
Sampling Cache	4096 sets × 6 ways. Each entry: 32 bits tag, 2 bits status, 32 bits IP.	199
Per-IP Statistics	128 sets × 8 ways. Each entry: 32 bits IP, 39 bits counters, 1 bit enabled.	9
Ongoing Requests	128 entries. Each entry: 1 bit valid, 32 bits tag, 12 bits timestamp, 1 bit demand.	0.7
Active cores counter	4 bits (1 per core).	<0.01
LLC Total		**209.7**

Table 1: Storage requirements of BertiGO

differentiated per core and require a cache-like structure to store the counters for each IP. A special IP is the 0, which indicates that the LLC request is generated from the L1D or L2 prefetcher.

After a number of evictions, that is, when the eviction counter saturates, the useless and useful counters are checked. Only if the useful counter multiplies the useless counter by a large number, the next-line prefetching is enabled (globally or per IP). After the decision, the three counters are multiplied by 0.75 in order to continue learning until the next saturation. We disable ANeLin entirely when bandwidth utilization is high, as next-line prefetching at the LLC would further saturate memory. Alternatively, the useful-to-useless threshold can be raised dynamically under bandwidth pressure, making the prefetcher more conservative rather than completely disabled.

5 The Multicore Case

For multicore configurations, we selectively disable some features. In shared caches, disabling the filter allows prefetch requests to serve as hints to L1 and L2 that an entry is soon to be reused. Additionally, without a filter, cores that prefetch aggressively are naturally penalized when they pollute the shared cache, creating an implicit feedback mechanism. We use the LLC to signal BertiGO when multiple cores are active, triggering filter bypass.

Multicore prefetching introduces complex interactions that may benefit from replacement policies or dynamic adaptation techniques [2]. We defer comprehensive multicore optimization to future work and focus on maximizing single-core performance within the competition scope. Accordingly, we also disable ANeLin and set-dueling in multicore configurations to avoid potential interference without thorough validation.

6 Implementation

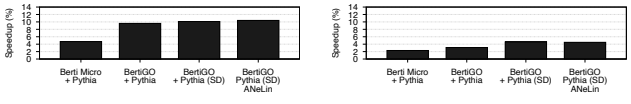
Table 1 details the storage requirements for each component. The L1D prefetcher uses 29.75 KB, dominated by the Region-based prefetch filter (15.6 KB) and History Table (10.52 KB). The L2 prefetcher uses the baseline Pythia with set dueling. The LLC prefetcher uses 209 KB for its sampling cache and per-IP statistics.

L1D Prefetcher. Following the original Berti design, we measure fetch latency for both demand misses and prefetch requests. Fetch latency is 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 computed by subtracting the stored timestamp from the current cycle. Table 1 details the full storage breakdown: the Region-based prefetch filter dominates at 15.61 KB, followed by the History Table at 10.53 KB, and the Delta Table at 2.03 KB. The IP-path history adds 32 bytes. Total overhead for latency tracking (Cache, MSHR, PQ extensions) is 1.55 KB. Note that all structures are scaled to fit the 32 KB budget of DPC-4.

Our prefetcher uses the provided cache interface to monitor the occupancy and size of the L1D PQ, RQ, and MSHR. To emulate Berti's latency measurement mechanism, we maintain shadow structures that mimic the L1D cache, PQ, and MSHR. These shadow structures are used solely for latency calculation; at no point do we use them to check if a cache line is already present in the cache. This approach is consistent with the original Berti MICRO 2022 artifact [4] and introduces no new behavior.

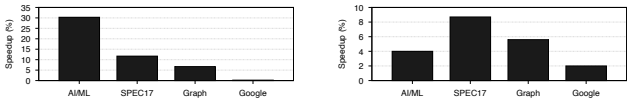
L2 Prefetcher. We use the original Pythia implementation (25.5 KB). The set-dueling mechanism adds 10 counters (5 candidates × 2 counters each, 64 bits each) plus auxiliary state (winner, phase) that accounts for 10 bytes. The overhead is 90 bytes.

LLC. ANeLin uses 209 KB. It includes 199 KB for the sampling cache (4096 sets × 6 ways), 9 KB for per-IP statistics (128 sets × 8 ways per core), and 0.7 KB for ongoing request tracking for latency computations. This storage is extended to maximize accuracy tracking; comparable performance can be achieved by sampling just a few sets. We also maintain a negligible 4-bit counter to track active cores for multicore detection.



(a) Full bandwidth (b) Limited bandwidth

Figure 1: Overall speedup over baseline.



(a) Full bandwidth (b) Limited bandwidth

Figure 2: Per-workload speedup breakdown.

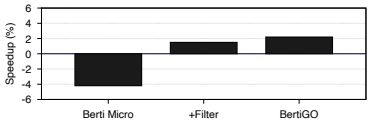


Figure 3: L1D ablation study showing filter and IP-path contributions with no L2 and LLC prefetcher on Full BW scenario over baseline.

7 Evaluation

Figure 1 shows the overall speedup over the DPC-4 baseline for both full and limited bandwidth configurations.

Our starting point is the MICRO 2022 Berti prefetcher, scaled to competition size and combined with Pythia, which achieves 4.65% geomean speedup under full bandwidth and 2.33% under limited bandwidth. By replacing part of the history table with our Region-based filter and adding IP-path signatures, BertiGO pushes this to 9.6% under full bandwidth and 3.06% under limited bandwidth. Set-dueling contributes an additional 0.6 percentage points under full bandwidth. Notably, set-dueling is even more effective under limited bandwidth, adding 1.5 percentage points by disabling Pythia on workloads where it consumes bandwidth without benefit. ANeLin contributes 0.2 percentage points under full bandwidth. The final configuration achieves 10.4% under full bandwidth and 4.5% under limited bandwidth.

Figure 2 breaks down performance by workload category. AI/ML workloads under full bandwidth benefit the most, reaching 30.3% speedup. Under limited bandwidth, they achieve 4.0%. SPEC17 benchmarks achieve 11.7% under full bandwidth and 8.7% under limited bandwidth. Graph workloads achieve 6.7% and 5.7%, and Google traces 0.2% and 2.0%, respectively.

Figure 3 isolates the L1D contributions under full bandwidth, without any L2 or LLC prefetcher. In this configuration, Berti Micro hurts performance by 4.1% compared to the baseline with Berti DPC-3 and Pythia. The Region-based filter reaches a speedup of +1.6% over baseline. IP-path signatures add another 0.7 percentage points for a total of +2.3%.

Figure 4 summarizes the competition score. Our submission achieves 10.4% on full bandwidth and 4.5% on limited bandwidth.

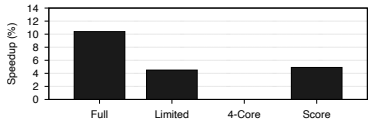


Figure 4: Summary: Full BW (10.4%), Limited BW (4.5%), 4-Core (0%), Score (4.9%).

In a randomized subset of mixed traces for multicore, we achieve no speedup over baseline. The geometric mean score is 4.9%.

8 Conclusions

We have presented BertiGO, an enhanced L1D prefetcher that addresses two key inefficiencies of the original Berti design.

First, the region-based prefetch filter eliminates redundant prefetch requests by tracking recently accessed lines within each page. This allows Berti to allocate resources more effectively, and for L2 prefetches, it additionally learns to suppress requests for lines that are never used. This configuration alone, without L2 or LLC prefetching, outperforms the baseline with Berti DPC-3 and Pythia by 2.3%.

Second, IP-path signatures augment the instruction pointer with execution context, enabling Berti to distinguish the same instruction reached via different paths. Combined with the filter, BertiGO achieves 9.6% speedup when paired with Pythia at L2, nearly doubling the Berti Micro result of 4.7%.

Beyond improving the L1D prefetcher, we show that dynamic policies can outperform static ones across the cache hierarchy. At L2, set-dueling lets each workload choose its preferred Pythia feature policy. This is especially effective under limited bandwidth, where it adds 1.5 percentage points by disabling Pythia on workloads where prefetching hurts. At LLC, adaptive next-line prefetching contributes additional gains, mostly on SPEC workloads.

On the DPC-4 training traces, our submission achieves 10.4% speedup under full bandwidth and 4.5% under limited bandwidth, with AI/ML workloads reaching 30.3%.

9 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] 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.

[2] Charles Block, Gerasimos Gerogiannis, and Josep Torrellas. 2025. Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture®*. 884–898.

[3] Thomas Hurst. [n. d.]. Bloom Filter Calculator. https://hur.st/bloomfilter/. Accessed: 2024-12-24.

[4] 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.

[5] Seth H. Pugsley, Zeshan Chishti, Chris Wilkerson, Peng fei Chuang, Robert L. Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers. In *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 626–637.

[6] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2008. Set-dueling-controlled adaptive insertion for high-performance caching. *IEEE micro* 28, 1 (2008), 91–98.

[7] Alberto Ros. 2019. Berti: A Per-Page Best-Request-Time Delta Prefetcher. In *The 3rd Data Prefetching Championship*.

[8] Alberto Ros and Alexandra Jimborean. 2023. Wrong-path-aware entangling instruction prefetcher. *IEEE Trans. Comput.* 73, 2 (2023), 548–559.