Emender: Optimizing Prefetch Priority and Throttling in VBerti+Pythia

Jiajie Chen
Tsinghua University
Beijing, China
cjj21@mails.tsinghua.edu.cn

Tingji Zhang
Tsinghua University
Beijing, China
ztj22@mails.tsinghua.edu.cn

Xiaoyi Liu
Tsinghua University
Beijing, China
xiaoyi-l23@mails.tsinghua.edu.cn

Xuefeng Zhang
Tsinghua University
Beijing, China
zhang-xf21@mails.tsinghua.edu.cn

Peng Qu
Tsinghua University
Beijing, China
qp2018@tsinghua.edu.cn

Youhui Zhang
Tsinghua University
Beijing, China
zyh02@tsinghua.edu.cn

Abstract

This paper proposes optimizations to the state-of-the-art VBerti + Pythia data prefetchers, focusing on prefetch prioritization and throttling through four key improvements: a Pending Target Buffer to prioritize high-priority prefetch requests; a hardware-efficient Cuckoo Filter to eliminate duplicate requests for cached data; a Dynamic Confidence Threshold that adjusts prefetching aggressiveness based on load instruction miss rates; and an L3 Fairness-based Throttling mechanism that regulates per-core prefetching using L1D/L2 cache feedback. Evaluation on DPC4-Traces demonstrates speedups of 6.6% (1C.fullBW), 2.0% (1C.limitBW), and 6.1% (4C), yielding an overall performance improvement of 4.8%.

1 Introduction

A data prefetcher is a hardware or software component designed to enhance computer system performance. Its primary function is to predict data that the processor is likely to access in the near future and proactively load it from slower main memory into faster cache memory, thereby reducing or hiding data access latency. As the performance gap between processor and memory speeds, often termed the memory wall, continues to widen, data prefetching has grown increasingly critical. By analyzing runtime memory access patterns, such as sequential, stride, or more complex behaviors, prefetchers attempt to predict subsequent memory addresses and issue prefetch requests. In an ideal scenario, when the processor actually requires the data, it already resides in the cache, thus avoiding expensive cache miss penalties.

Based on our evaluation of existing data prefetchers, we found that combining VBerti [11] as the L1D cache prefetcher with Pythia [2] as the L2 cache prefetcher yields the best performance. To differentiate the MICRO-55 Berti prefetcher [11] from the DPC-3 Berti prefetcher [13], we refer to the former as VBerti, where the "V" denotes virtual addressing. Unlike Berti, which finds per-page timely deltas and does not prefetch across page boundaries, VBerti tracks timely deltas by instruction pointer (IP) and enables cross virtual page prefetches. Further analysis revealed several limitations in VBerti, leading us to propose the following improvements:
(1) VBerti only orders prefetch requests generated by the same load instruction. In practice, multiple load instructions can trigger prefetches concurrently, and ordering prefetches across different instructions could improve overall prefetching performance. To address this, we implemented a Pending Target Buffer that prioritizes prefetch requests with higher confidence across multiple load instructions.
(2) A subset of generated prefetch requests hits the L1D cache, yet these still occupy space in the Prefetch Queue. By filtering out such requests before they enter the queue, more useful prefetches can be accommodated. However, performing early L1D cache lookups for every prefetch request is challenging in hardware. Instead, we employ a Cuckoo Filter [8] to track cache line addresses present in the cache, thereby filtering redundant prefetch requests efficiently.
(3) We note that in certain applications, VBerti generates a substantial number of ineffective prefetch requests for loads that are difficult to predict. To address this, we employ a saturating counter to track the miss rate per load instruction and dynamically adjust the confidence threshold, a mechanism we refer to as Dynamic Confidence Threshold. This enables the prefetcher to reduce its activity in scenarios where prefetching is challenging.
(4) While the VBerti+Pythia combination achieves strong single-core performance, its aggressive prefetching can induce resource contention in multi-core settings, resulting in starvation on some cores. To enhance fairness, we utilized metadata exchange between prefetchers to allow the L3 cache prefetcher to regulate bandwidth allocation, an approach termed L3 Fairness-based Throttling. This ensures balanced L3 bandwidth utilization across cores and improves Harmonic Speedup [7].

Through ablation studies, we tested and found that the four improvements: Pending Target Buffer, Cuckoo Filter, Dynamic Confidence Threshold, and L3 Fairness-based Throttling, individually contributed to total score improvements of 0.2%, 0.8%, 0.1%, and 0.9%, respectively. By integrating these enhancements, an overall improvement of 2.4% in the total score was achieved on top of VBerti+Pythia. Compared to the baseline of Berti+Pythia, this resulted in speedups of 6.6% (1C.fullBW), 2.0% (1C.limitBW), and 6.1% (4C), yielding an overall performance improvement of 4.8%.

2 Related Work

The Offset Prefetcher is a category of prefetchers whose behavior is as follows: when accessing a cache line at address X, it prefetches the cache line at address X+O, where O is the offset, which can be fixed or dynamically learned.

The Best Offset Prefetcher [9] is an Offset Prefetcher based on the idea that the optimal offset may differ across programs; thus, the

best offset needs to be computed dynamically. The algorithm is as follows: Record the base address Y-O of recently prefetched cache lines in a Recent Requests table, where Y is the prefetched cache line address and O is the prefetch offset. When a cache line with address X experiences a cache miss or is accessed as a prefetched hit, the prefetcher evaluates a predefined set of offsets O1, O2, ..., On by computing X-Oi. If the result exists in the Recent Requests table, the corresponding score for Oi is incremented. When any offset reaches a score threshold or a time limit expires, the offset with the highest current score is selected for prefetching. The scoring mechanism then resets, and learning restarts. Compared to a fixed-offset Offset Prefetcher, the Best Offset Prefetcher can dynamically identify the offset most suitable for the current application.

VBerti advances further upon the Best Offset Prefetcher. While the Best Offset Prefetcher posits that the optimal offset varies across programs and needs to be found dynamically, VBerti argues that even different load instructions within a program may have different optimal offsets, necessitating distinct optimal offsets for different load instructions. Unlike the Best Offset Prefetcher, VBerti does not rely on a predefined list of offsets. Instead, it determines suitable offsets based on actual cache line addresses. The specific implementation is as follows: First, VBerti maintains separate access histories for different load instruction PCs in History Table. Next, VBerti measures L1D cache miss latency, the time from a miss generation to data return. To ensure timely data prefetching, the prefetch time plus cache miss latency must be earlier than the actual usage time. As an Offset Prefetcher, to prefetch a cache line at address Y accessed by a specific load instruction, VBerti needs to find an earlier cache line at address X accessed by the same load instruction, such that prefetching Y when accessing X is timely. It then computes the difference Y - X as a suitable offset. This confirms that using this offset for prefetching is reasonable, and its corresponding score is increased. Finally, offsets with high scores are used for prefetching.

Pythia is an Offset Prefetcher based on Reinforcement Learning (RL). The concept of RL involves an agent perceiving a state, outputting an action, and receiving a reward, with the goal of maximizing long-term rewards. Here, the action is determining the offset used by the Offset Prefetcher. The Q-function takes the state and action as input and outputs a predicted reward value. Thus, RL aims to find an accurate approximation of the Q-function and then select actions based on the learned Q-function. Rewards are assigned based on the prefetcher's effectiveness: (1) Accurate and timely: Prefetched data is accessed after being loaded into the cache. (2) Accurate but late: Prefetched data is accessed before the prefetch completes, indicating accuracy but insufficient timeliness. (3) Loss of coverage: The prefetched address crosses a page boundary (since physical addresses are used), causing the prefetch to fail. (4) Inaccurate: Prefetched data is never accessed, indicating incorrect prefetching. (5) No-prefetch: No prefetch action is taken.

Pythia maintains prefetch history using an Evaluation Queue (EQ). When a core accesses a cache line, it checks if the cache line's address exists in the EQ. If it does, and the cache line hits, an Accurate and timely reward is given. If the address is in the EQ but the cache line misses, an Accurate but late reward is given. During prefetching, based on the current state, the action yielding the highest corresponding Q-value is selected (with a small probability of choosing a random action). This chosen action is used as the offset for prefetching, and the information for this prefetch is then inserted into the EQ. If the action equals 0, meaning no data is to be prefetched, a No prefetch reward is given. If the address to be prefetched crosses a physical page boundary, a Loss of coverage reward is given.

3 Design

3.1 Emender at L1D

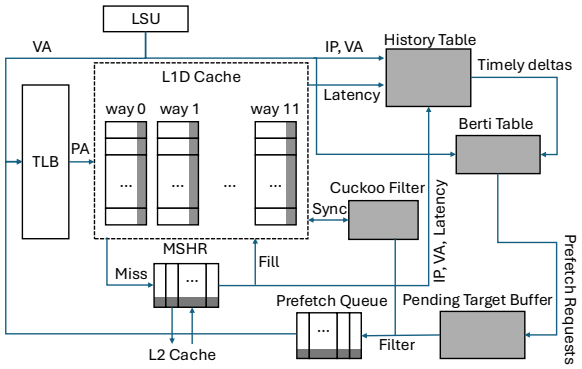

Figure 1: Design of Emender-L1D cache, adapted from Figure 5 of VBerti [11]. Hardware changes are denoted in gray.

Emender at L1D (abbreviated as Emender-L1D) builds upon VBerti by introducing a Pending Target Buffer, a Cuckoo Filter, and modifications for Dynamic Confidence Threshold and L3 Fairness-based Throttling. Its overall architecture is illustrated in Figure 1. The following sections detail its key components.

The prefetcher employs a History Table to record access patterns. Organized as a multi-way set-associative structure, it can store multiple histories for the same program counter (PC). Each entry contains a tag derived from hashing the load instruction's PC, the address of an accessed cache line, and a timestamp. During a query, a time difference is calculated by subtracting the cache miss latency from the current timestamp. Entries with timestamps earlier than this difference are identified, and their corresponding cache line addresses are used for subsequent offset calculations.

Concurrently, the prefetcher maintains a multi-way fully associative Berti Table, which tracks the states of multiple optimal offsets for different load instructions. Each Berti Table entry includes:

- A tag derived from hashing the load instruction's PC.
- A counter recording the number of updates to this entry, serving as a confidence metric.
- A miss counter characterizing the miss rate of the corresponding load instruction.
- Several delta entries, each consisting of: The delta value (the offset). A confidence value representing the proportion of prefetches this offset can cover. A status indicator for whether prefetching is enabled for this offset.

The newly introduced miss counter enables Dynamic Confidence Threshold: a high miss rate triggers throttling. Entries with very high confidence remain unchanged, while those with moderate

confidence previously eligible for L1D prefetching, may now be prefetched only to L2. Deltas with lower confidence may be downgraded from L2 prefetching to none. To avoid excessive throttling during learning phases, the counter saturates and is initialized to its minimum. It increments on a cache miss and decrements probabilistically on a hit. Thus, a persistently elevated miss rate raises the counter's expected value above zero, signaling sustained high miss activity.

To track memory access latency, timestamps are stored with each MSHR and Prefetch Queue entry. To comply with DPC4 rules, a fully associative Latency Table (also not shown in figure) records each inflight request's address, hashed PC, start time, and whether it originated from a demand miss or prefetch.

On a hit to a prefetched line, the Berti Table is updated using the latency previously stored. This was achieved by storing latencies directly in the cache. To comply with DPC4 rules, we added a Shadow Cache (not shown in figure) that records tags and latencies without storing data.

When a cache line is loaded, Emender-L1D searches the History Table for earlier matching entries, computes address deltas, and updates the Berti Table: the entry's counter is incremented, and matching deltas have their coverage increased. Once an entry's update count reaches a threshold, Emender-L1D evaluates its deltas: high coverage triggers L1D prefetching, medium coverage triggers L2 prefetching, and low coverage results in no prefetch.

We observe that VBerti issues prefetches in confidence order, but high-confidence requests are often dropped when the Prefetch Queue is full. To address this, Emender-L1D logs prefetch requests into a Pending Target Buffer, which globally sorts them by confidence and issues prefetches in descending order. Each entry stores the address, confidence, status, hashed IP, sequence ID, and failure count. On insertion, duplicates are filtered and the lowest-confidence entry is replaced if full. Prefetches from older instructions or with excessive failures are evicted periodically.

Additionally, we observed that VBerti generates many redundant prefetches for already cached lines. Therefore, we introduced the hardware friendly Cuckoo Filter, a probabilistic data structure used to maintain which virtual addresses of cache lines are stored in the L1D cache. If the L1D cache uses VIVT, no additional information needs to be saved: the Cuckoo Filter can be maintained directly with virtual addresses. If the L1D cache uses VIPT, additional hash for the Cuckoo Filter is stored along with the cache line. In our implementation, this additional information is also accounted for in the capacity of the Shadow Cache. To reduce hardware overhead, the MaxNumKicks parameter of Cuckoo Filter is set to two.

Finally, to support L3 Fairness-based Throttling across cores, the throttle configuration is read from Emender-L3 based on metadata exchanged between prefetchers. When throttled, Emender-L1D stops prefetching.

3.2 Emender at L2

Emender employs the largely unmodified Pythia prefetcher for the L2 cache. As with the VBerti adaptation, we adjusted Pythia to respond to fairness-based throttling at L3. First, while Pythia already incorporates throttling for high memory-bandwidth situations, we extended it to react to L3 throttling requests, thereby adjusting its reward values accordingly.

Furthermore, to help the L3 cache prefetcher decide which cores to throttle, the L2 cache prefetcher periodically collects statistics on useless prefetch requests, the cache lines that are prefetched but evicted without ever being accessed. These statistics, already maintained by the DPC4 ChampSim framework, are communicated to L3 via inter-prefetcher metadata exchange. Simultaneously, it listens for metadata broadcasts from L3, updates its own throttling state, and forwards the metadata to Emender-L1D.

3.3 Emender at L3

Emender does not perform actual prefetching at the L3 cache. Our experiments show that the access information available at L3 is difficult to exploit for effective prefetching. Although more aggressive prefetching could improve single-core performance, it generally performs poorly in multi-core, bandwidth-constrained scenarios. Moreover, under the DPC4 scoring methodology, adding an L3 prefetcher almost always lowers the overall score. Therefore, we omit prefetching at L3 and instead implement an L3 Fairness-based Throttling mechanism, drawing inspiration from [4] but adopting a simpler design. Specifically, each L2 cache prefetcher reports to L3 the count of useless prefetches per core. The L3 module aggregates these counts and throttles the core with the highest number of useless prefetches while another core has less.

3.4 Storage Overhead

As per the competition requirements, the prefetchers deployed at the L1D cache, L2 cache, and L3 cache must not exceed capacities of 32KB, 128KB, and 256KB, respectively. The Table 1 details the capacity usage of our prefetchers at each cache level, demonstrating that they all comply with the requirements.

4 Simulation results

We evaluated our design on the DPC4 ChampSim codebase under three configurations specified by the competition rules: (1) 1C.fullBW: single-core with 4800 MT/s memory bandwidth; (2) 1C.limitBW: single-core with constrained 800 MT/s bandwidth; and (3) 4C: four-core with a quadrupled last-level cache (LLC).

Experiments were conducted using the publicly available DPC4-Traces. For single-core tests, each trace was run individually and results were aggregated using the geometric mean of IPC. For multi-core evaluation, we randomly assigned one trace per core, computed the harmonic speedup relative to baseline, and then took the geometric mean of these speedup values. The final competition score is the geometric mean of the multi-core result with the scores from 1C.fullBW and 1C.limitBW.

Due to time constraints, we report results using a subset (1% instructions of each trace) of the full trace set; however, preliminary tests indicate that overall trends remain consistent, for instance, the 1C.fullBW profile achieves a 6.6% speedup with the subset versus 6.5% with the full set. Besides comparing Emender against the DPC4 competition baseline (Berti+Pythia), we also evaluated configurations with no prefetcher and with several different L1D prefetchers: Bingo [1], Gaze [6], and VBerti [11].

Table 1: Storage Overhead of Emender

Component of L1	Storage Overhead
Latency Table	40 entries, each entry contains: 58-bit cache line address, 16-bit tag, 16-bit time, 1-bit prefetch indicator, 3640 bits in total
Shadow Cache	64 set x 12 way, each entry contains: 58-bit cache line address, 12-bit latency, 22-bit index+fingerprint for Cuckoo Filter, 1-bit prefetch indicator, 71424 bits in total
History Table	16 set x 32 way, each entry contains: 16-bit tag, 24-bit address, 16-bit time; 5-bit FIFO replacement policy in each set, 28752 bits in total
Berti Table	64 entries, each entry contains: 16 deltas (5-bit confidence, 13-bit delta, 2 bit replacement policy), 5-bit confidence, 10-bit miss confidence, 16-bit tag; 6-bit extra FIFO replacement policy, 22470 bits in total
Pending Target Buffer	80 entries, each entry contains: 58-bit cache line address, 5-bit confidence, 2-bit replacement policy, 16-bit tag, 4-bit load sequence, 1-bit trial counter; 4-bit extra load sequence counter, 6884 bits in total
Cuckoo Filter	1024 set x 4 way, each entry contains: 1-bit valid, 12-bit fingerprint, 53248 bits in total
Emender at L1D Total	186418 bits (22.8 KiB)
Emender at L2	Same as Pythia (25.5 KB)
Emender at L3	Uselessness counters (64 B)

As shown in Figure 2, VBerti slightly outperforms other prefetchers, while Emender builds on VBerti to achieve higher single- and multi-core scores.

Figure 3 presents the S-curve of Emender versus Berti+Pythia under the 1C.fullBW profile, with test cases sorted by speedup ratio. Performance degradation in the worst case remains below 10%, whereas the best case shows a speedup exceeding 1.4x.

Through ablation studies, we tested the performance improvement of Emender compared to versions with individual features removed, as shown in Table 2. As can be observed, both the Cuckoo Filter and L3 Fairness-based Throttling deliver notable performance improvements, with the latter contributing the most significant improvement in 4C scenarios. In contrast, the improvements brought by the Dynamic Confidence Threshold and the Pending Target Buffer are relatively modest.

Regarding the Cuckoo Filter employed for prefetch filtering, it is worth noting that as a probabilistic data structure, it may produce false positives. In our experiments, we measured the error rate and found it to be negligible at only 0.03%.

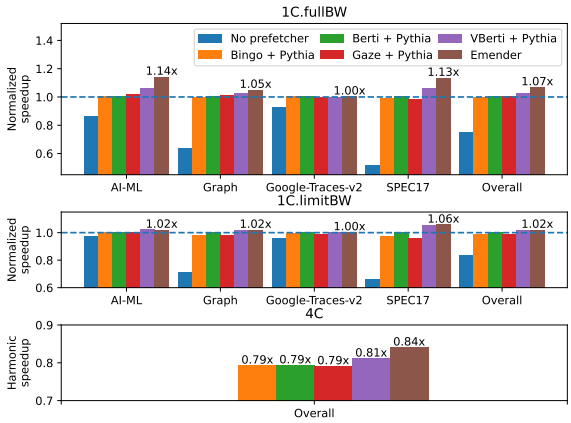

Figure 2: Performance of Emender versus other prefetchers

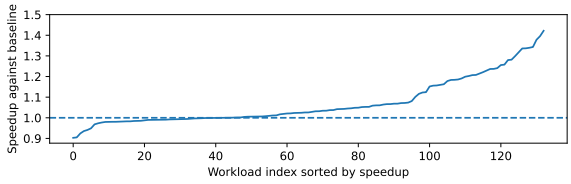

Figure 3: S-Curve of Emender versus baseline in 1C.fullBW profile

Table 2: Performance impact of disabling individual features

Feature	1C.fullBW	1C.limitBW	4C	Overall
Cuckoo Filter	**1.3%**	**0.2%**	1.0%	0.8%
L3 Fairness-based Throttling	0.0%	0.0%	**2.6%**	**0.9%**
Dynamic Confidence Threshold	0.1%	**0.2%**	0.2%	0.1%
Pending Target Buffer	0.4%	0.0%	0.4%	0.2%

5 Discussion and Future Work

Berti Table Implementability. In the submitted version, the Berti Table employs a fully-associative design, as the competition imposed no hardware constraints on timing. However, considering practical hardware implementability, a set-associative organization could be adopted. Future work may include evaluating this design, for instance, following the approach in Berto [12] by converting the fully-associative table to a 16-set, 4-way configuration with the NRU (not recently used) replacement algorithm.

Hardware Design of Pending Target Buffer. Since the Pending Target Buffer requires sorting, actual hardware implementations may partition entries into multiple buckets based on priority, employing a bucket sort-like mechanism to reduce sorting overhead. However, this approach may introduce performance impact that require further investigation.

Comparison of Prefetch Filtering Techniques. Prior studies have explored alternative filtering approaches, for instance, T-SKID [10] uses Shadow Cache and Prefetch Queue for filtering; Sangam [5] employ a 40-entry Recent Access Filter to track the most recent access or prefetched addresses; PPF [3] utilizes a perceptron to determine whether a prefetch should be filtered. Emender employs multiple layers of prefetch filtering: the Cuckoo Filter eliminates redundant prefetches for cache lines already present in L1D, while the Pending Target Buffer and Prefetch Queue perform deduplication. A comprehensive comparison of these different filtering techniques, examining their trade-offs in terms of hardware overhead, accuracy, and performance impact, remains a promising direction for future research.

6 Conclusion

Building upon the state-of-the-art VBerti+Pythia, we designed Emender, a refined prefetcher incorporating several key enhancements: (1) a Pending Target Buffer to refine prefetch prioritization; (2) a Cuckoo Filter to eliminate redundant prefetches for already-cached data; (3) a Dynamic Confidence Threshold that adjusts prefetch aggressiveness based on observed miss rates; and (4) an L3 Fairness-based Throttling to improve performance under multicore contention. Collectively, these optimizations yield a performance gain of 6.6% in the 1C.fullBW profile, 2.0% in 1C.limitBW, 6.1% in the 4C multicore profile, and an overall score improvement of 4.8% compared to the Berti+Pythia baseline.

Acknowledgments

This work was supported in part by the Brain Science and Brain-like Intelligence Technology-National Science and Technology Major Project under Grant No. 2025ZD0215500, Huawei Technologies Co., Ltd. and the Huawei-Tsinghua Joint Lab on Computer Architecture, the National Natural Science Foundation of China under Grant No. 62250006, the Tsinghua University Initiative Scientific Research Program under Grant No. 2022Z11ZRB002, and the Suzhou-Tsinghua Innovation Leadership Program under Grant No. 20222002100. The authors sincerely appreciate the valuable feedback provided by the anonymous reviewers.

References

[1] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Accurately and maximally prefetching spatial data access patterns with bingo. *The Third Data Prefetching Championship* (2019).

[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 *MICRO-54*. 1121–1137.

[3] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V Gratz, and Daniel A Jiménez. 2019. Perceptron-based prefetch filtering. In *Proceedings of the 46th International Symposium on Computer Architecture*. 1–13.

[4] Charles Block, Gerasimos Gerogiannis, and Josep Torrellas. 2025. Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching. In *MICRO-58*. 884–898.

[5] Mainak Chaudhuri and Nayan Deshmukh. 2019. Sangam: A multi-component core cache prefetcher. *3rd Data Prefetching Championship* (2019).

[6] 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 *HPCA '25*. IEEE, 173–187.

[7] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO-42*. 316–326.

[8] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[9] Pierre Michaud. 2015. A best-offset prefetcher. In *2nd Data Prefetching Championship*.

[10] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. 2019. T-SKID: Timing Skid Prefetcher. *3rd Data Prefetching Championship* (2019).

[11] 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 *MICRO-55*. IEEE, 975–991.

[12] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibánez, Víctor Viñnals-Yúfera, and Alberto Ros. 2025. A complexity-effective local delta prefetcher. *IEEE Trans. Comput.* (2025).

[13] Alberto Ros. 2019. Berti: A per-page best-request-time delta prefetcher. *The 3rd Data Prefetching Championship* (2019).

A Simulation Results on Evaluation Traces

For the camera-ready submission, we evaluated our prefetcher on the official evaluation traces (DPC4-All-Traces), which differ from the training traces (DPC4-Traces) used in the main text. Unlike our earlier experiments that utilized a 1% instruction subset, we tested the complete evaluation traces. The results are presented in Figure 4.

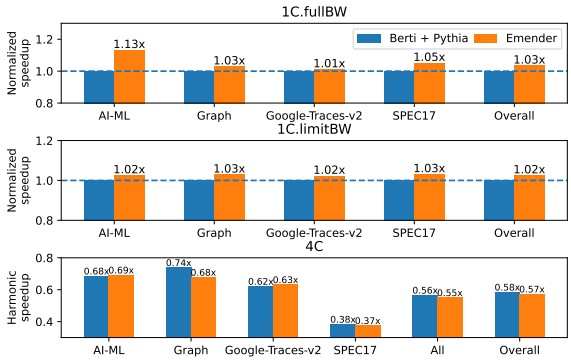

Figure 4: Performance of Emender versus baseline on evaluation traces

The single-core performance trends remain largely consistent with our training trace results, with only the SPEC17 category exhibiting a slight regression. This decline can be attributed to the inclusion of additional benchmarks in the evaluation trace whose primary bottlenecks lie outside memory access (e.g., leela). More significantly, the overall score reduction is driven by the substantially increased proportion of Google-Traces-v2 in the evaluation trace (rising from 32% to 59%), for which our optimizations provide minimal benefits. Despite this, when examining individual traces, the peak performance improvement is notably higher, achieving up to 1.71x IPC speedup on specific traces.

The multi-core performance exhibits divergent characteristics compared to the training trace results. Our preliminary experiments were conducted exclusively on the All category, where random sampling across all benchmark types generated greater trace diversity, thereby amplifying the effectiveness of our throttling strategy. In contrast, the evaluation trace performs sampling independently within each category, resulting in more homogeneous behavior across cores. Under these conditions, the current throttling mechanism proves suboptimal.