Toppy: A Global Offset-based Data Prefetcher for Instruction Heavy Server Workloads

Naman Sharma, Biswabandan Panda
Indian Institute of Technology Bombay

***Abstract*—Hardware data prefetching is a well-established technique for mitigating the long latency of off-chip memory accesses. However, there is little focus on data prefetching techniques specifically designed for server workloads with huge instruction footprints (10s of MBs). As the instruction footprint of server workloads is higher than that of workloads like SPEC CPU2017, even to achieve decent performance gains, substantial metadata storage is needed, especially for prefetchers that use the program counter (PC) to understand memory access patterns. On top of that, server workloads exhibit irregular memory access patterns as control flow jumps between user and kernel level accesses. To address these issues, we propose Toppy, a global offset-based hardware prefetching technique tailored for server workloads that does not use PC for data prefetching. Toppy uses the Misra-Gries algorithm to select the top-k persistent global offsets that persist for the entire program and prefetch based on those offsets. Compared to an L1D IP-stride prefetcher, Toppy provides a performance improvement of 9.22%, whereas the next-best Bingo prefetcher delivers a performance boost of 6.28%. Toppy's performance comes with a total storage budget of 274 bytes, whereas Bingo requires 119 KB.**

***Index Terms*—Data prefetching, performance.**

I. Introduction

Hardware data prefetching is a well-established technique for mitigating the long latency of off-chip memory accesses. Although several hardware data prefetchers [1]–[6] have been recently proposed, there has been little focus on prefetching techniques designed explicitly for instruction-heavy server workloads [7]. Figure 1 shows the performance of various hardware prefetchers for memory-intensive (LLC misses per kilo instructions > 1) server workloads normalized to an IP-stride prefetcher at the L1D. Most of the prefetchers provide a performance improvement of less than 2%. Bingo [4] is the best prefetcher with a performance improvement of 6.28%. We utilize the best-tuned versions of all the prefetchers provided by their respective authors.

A high-performing combination of prefetchers used by the winner of the recently concluded 4th data prefetching championship (DPC-4) [8] also fails to outperform Bingo for these workloads. In terms of prefetch accuracy, even highly accurate prefetchers like Berti, which achieves close to 90% accuracy with SPEC CPU2017 and GAP workloads, deliver 13.73% accuracy for these server workloads.

Huge instruction footprints and PC-based prefetchers. We find that for server workloads that we study, on average, there are 552.94 unique PCs for every one million instructions, with a maximum of 1479.81 per one million for Google workloads. Whereas for SPEC CPU2017, we see an average of 6.80 unique PCs per one million. One can argue that increasing the prefetch table sizes of PC-based prefetchers will improve performance. However, it is not the case. We observe marginal performance improvements with the increase in prefetch table size (6.28% becomes 7.22% with an increase in table size from 119KB to 476KB per core). We observe similar trends for Berti, where a 10X increase in the Berti table improves the performance by 0.5%.

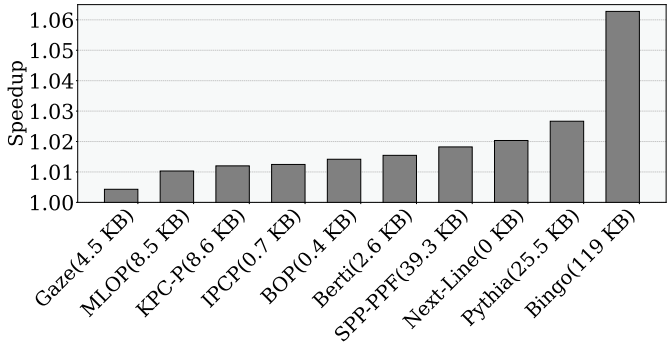

Fig. 1. Performance of hardware prefetchers for memory-intensive server workloads selected from CloudSuite [9], CVP1 [10], XSBench [11], Google [12], and Java Renaissance [13] normalized to an L1D IP-stride.

Ineffective non-PC-based prefetchers. Kill the program counter for prefetching (KPC-P) [14] is one of the prefetchers that do not use PC and can be a good alternative. However, on average, KPC-P performs similar as Berti for these server workloads, with an average performance improvement of 1.2%. One of the primary reasons for this behavior is the irregular deltas. The other non-PC-based prefetchers, like BOP [1] and MLOP [15], are also ineffective (Figure 1).

Key observations. We observe that a small fraction of global (not specific to PCs) offsets covers a good fraction of cache accesses. An offset is the difference between two cache line-aligned addresses. Global offsets are offsets that are not specific to any PC or memory region and follow the cache access order. We observe that, on average, approximately ≈50% of LLC accesses can be covered if we can prefetch based on the top-10 global offsets. We define Top-k offsets as the k most frequent offsets observed at the LLC.

Our approach. We propose an L3 offset-based prefetcher named Toppy that uses the k most frequent offsets (O1, O2, ..., OK) for prefetching. For an access A, Toppy prefetches A+O1, A+O2, ... A+OK into LLC. In this way, Toppy does not correlate any specific PC or an access stream with an accessed address. Instead, it treats each address in isolation, and based on the frequency of each prefetch offset, it issues prefetch requests. Toppy uses an adaptive k based on offset accuracy and available DRAM bandwidth, providing a sweet

spot between performance and DRAM traffic. In contrast to prior offset-based prefetchers like BOP and MLOP that exploit *transient* offsets by evaluating offsets in an interval, Toppy learns the *persistent* offsets.

II. Why Yet Another Global Offset Prefetcher?

Irregular local offsets. State-of-the-art local offset prefetchers like Berti and Berto fail to predict the future offsets because of irregular offsets. Also, the offsets are big, not limited to small offsets within an OS page. This affects the learning of offsets. Bingo is the only per PC prefetcher that delivers good performance, thanks to its multiple signatures (PC, PC+address, and PC+offset).

Transient offsets. Global offset prefetchers like BOP and MLOP fail to deliver high performance gains as the global offsets selected by these prefetchers are *transient*. We define an offset as *transient* if it appears in one part of the program and disappears in the future. State-of-the-art global offset-based prefetchers use an evaluation epoch, where these techniques try to select the best offset(s). However, the best offset(s) do not recur in future epochs. Second, all the global offset prefetchers fail to select offsets in the evaluation rounds when none of the offsets clear the threshold for offset selection. On average, more than 15% of the evaluation rounds of MLOP fail to identify a single offset for prefetching.

Missed opportunities. Instead of evaluating offsets on the fly, we observe that offsets that *persist* for a long interval are good candidates for prefetching for these server workloads, and indirectly, it nullifies the effect of transient offsets, providing better prefetch coverage. A top-10 prefetcher that selects the 10 most frequent offsets can cover 50% of LLC accesses, and even top-5 offsets can cover more than 47.5% of LLC accesses. Note that these offsets recur multiple times and are mostly stable across different parts of the program. For example, for CloudSuite, offsets -8, -1, 1, 2, 4, 5, 9, 13, 42, 43, 60, 62 persist for 75.24% of phases, where one phase is of 1024 LLC misses. For Google workloads, offsets -12, -7, -6, -5, -4, -2, -1, 1, 2, 3, 4, 5, 7, and 12 persist for 75.10% of phases.

Challenges. Designing a persistent global offset prefetcher can cover a good fraction of LLC misses. However, there are a few challenges. Because server workload memory access patterns are irregular, achieving high prefetch accuracy and coverage is challenging. In fact, all state-of-the-art prefetchers (except Bingo) provide a prefetch accuracy of less than 40%. Second, the global offset prefetcher should be agile in identifying top-k offsets dynamically.

III. Toppy Data Prefetcher

We propose Toppy, an LLC prefetcher that prefetches based on frequent global offsets that persist for a long duration. A global offset is defined as the difference (at cache line granularity) between two consecutive accesses to LLC. Compared to prior works [1], Toppy does not pre-select a few offsets that it want to explore; instead, it explores all the possible offsets within an OS page. It monitors all LLC accesses and captures the top-k persistent offsets based on LLC accesses.

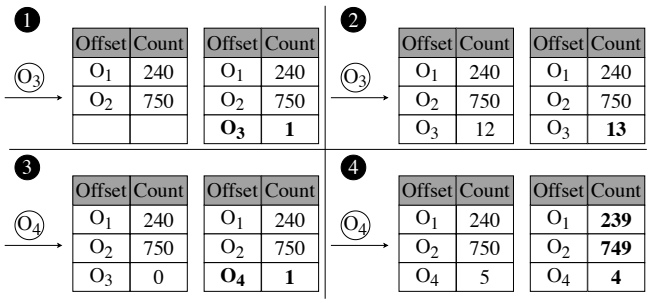

Fig. 2. Misra Gries algorithm in action for an offset table of size three. O1 to O4 are the offsets with their respective counts. Steps 1 to 4 show four possible scenarios as mentioned in Section III-A.

A. Design and implementation

Each time a load generates an LLC access, Toppy at LLC uses the top-k offsets from the LLC offset table to issue up to k prefetch requests. We use the Misra-Gries algorithm, which is designed to identify frequent elements in a stream while using bounded storage. It is easily implemented using a few counters. In fact, Misra–Gries uses k-1 counters to find top k, which is extremely lightweight in terms of implementation and storage cost. We do not use more refined ideas such as space-saving [16], which performs better than Misra-gries for skewed access streams. We employ this algorithm in our Toppy prefetcher to capture the most frequently occurring offsets at LLC. Figure 2 shows the steps involved in identifying top-k offsets with their frequencies (counts). On an LLC access, Toppy computes the global offset, and the offset goes through the following steps:

1. If the offset is not present in the offset table, a new entry is allocated, provided the table is not full.
2. If the offset already exists in the offset table, its corresponding counter is incremented.
3. If the table is full, an entry with a counter value of zero is replaced, and the new offset is filled and initialized with a counter value of one.
4. If neither 2 and 3 condition holds, the counters of all existing entries are decremented by one till one of the offsets reaches the counter value of zero.

Through this process, infrequent offsets are gradually replaced, while frequently occurring offsets maintain nonzero counters.

Toppy at the LLC in action. Upon a load access (A), the prefetcher computes the offset between the current memory address and the last accessed address (A_0). The computed offset ($A - A_0$) is recorded in the offset Table, which maintains all offsets in the range -63 to $+63$ along with their counts. The Offset table uses the Misra–Gries algorithm to identify the most frequent top-k offsets. The LLC prefetcher selects the top-k most frequent offsets $(O_1, O_2, \ldots, O_K)$. For each offset, it generates a prefetch address of the form. Finally, the generated prefetch requests are placed into the Prefetch Queue (PQ), from which they are issued to the DRAM. Note that duplicate requests are merged at the PQ, and the insertion order follows the Misra-Gries ranking.In case the PQ is full, we drop the prefetch requests.

Making Toppy adaptive to DRAM traffic. Toppy selects the top-k offsets, where the maximum value of k we set is 12.

TABLE I
OFFSET ACCURACY THRESHOLD BASED ON DRAM BANDWIDTH

Offset accuracy	DRAM bandwidth consumption
0.1	$< 25\%$
0.3	$\geq 25\%$ and $< 50\%$
0.4	$\geq 50\%$ and $< 75\%$
0.5	$\geq 75\%$

The size of the offset table should be chosen in a way such that the offset table does not become full, causing conflicts, leading to replacements. We also need to ensure that transient offsets do not replace an offset that would eventually become a persistent offset. A large value of k leads to an increase in DRAM traffic. It also affects the overall prefetch accuracy, as some of the offsets do not provide hits even if Misra Gries selects them. To balance prefetch accuracy and coverage, we extend Toppy and make it adaptive. With adaptive Toppy, LLC uses per offset prefetch accuracy to select top-k offsets.

The offset table tracks frequent offsets by storing their frequency count. We also use an accuracy tracking table to store the addresses generated by the Misra Gries and the offset used to generate the prefetch address. The issued and useful counters in the offset table are used for accuracy tracking, where issued records the number of predicted addresses for a given offset, and useful records the number of subsequent matches observed in the accuracy tracking table for those addresses. If a future demand load access (A) matches one of the prefetched addresses stored in the table, then the corresponding useful count for the offset used to generate that address is incremented in the offset table. At the same time, a new set of addresses generated using the top-k is also inserted into the accuracy tracking table, incrementing the issued count.

DRAM bandwidth conscious Toppy. To monitor the accuracy of each offset as selected by Misra Gries, we use an epoch (based on LLC misses) approach in which we check the offset accuracies of the top-k offsets selected by Misra Gries within that epoch. During that epoch, if the accuracy of an offset is higher than an accuracy threshold, then Toppy chooses that offset for prefetching. At the end of an epoch, Toppy resets all the counters related to issue and useful. So, at the beginning of an epoch, there are no offsets that cross the offset accuracy threshold. Note that the Misra-Gries algorithm continues learning global persistent offsets, and the predicted offsets are not removed at the end of the epoch. We only reset their corresponding counters. Resetting the issue and useful counters is necessary to filter out offsets that show high accuracy in the previous epoch but fail to generate accurate prefetch addresses in the current epoch.

Implementation details. To track the utility of Toppy and the available DRAM bandwidth, we use an epoch of size 1024 LLC misses, where we continue using top-k offsets from the previous epoch for the initial 128 LLC misses. So after every 1024 LLC misses, the prefetch issued, and the useful counters of the offset table are reset to zero. Based on DRAM bandwidth utilization, we set the lowest offset-accuracy threshold for top-k selection to 0.1 and to 0.5 under extreme DRAM bandwidth constraints. Table I shows the mapping of DRAM bandwidth consumption with the offset accuracy threshold. To track DRAM bandwidth, we use prior approaches used in DSPatch [17]. At the DRAM controller, we count the number of DRAM column access (CAS) commands for every $4\times$ tRC cycles. We scale these counts into 25%, 50%, and 75% of DRAM MTPS.

Storage overhead. Overall, the per-core Toppy at the LLC incurs a storage overhead of 274 bytes. Toppy uses an offset table of 16 entries and an accuracy tracking table of 128 entries. Each entry in the offset table stores a 7-bit offset, a 23-bit frequency counter, and a 10-bit counter for issues and useful prefetch requests. The accuracy tracking table is of two banks, each of 64 entries, where each entry stores a 12-bit tag and a 7-bit offset. We use a 10-bit counter for the evaluation window of 1024 LLC misses. Selected top-12 offsets are stored using 7-bit registers.

Why Toppy works? For server workloads, we observe frequent calls to libraries and kernel modules, driven by different control and data flows. Through Toppy, we indirectly capture the *frequent offsets* for each control flow and data flow, and in future calls to one of these libraries and modules, some of the *frequent* offsets cover the misses. In summary, global offsets capture locality at the global layout of the program, which includes data structure layouts within irregular server workloads. This locality is similar to temporal locality at the offset level, where some offsets recur, creating a few global offsets that repeat and hence show persistent behavior, which is captured by Toppy.

MLOP vs Toppy. MLOP [15] selects global offsets per lookahead, and it answers prefetching questions like which offset(s) is(are) best for different lookaheads. Toppy maintains a rank order, asking which global offsets are useful at the moment based on the global behavior of all offsets. Overall, Toppy is very close to MLOP. However, MLOP learns the local offsets again and again, keeping lookaheads in mind, whereas Toppy ranks the offsets that persist for a longer duration.

IV. EVALUATION

We use ChampSim [18], a trace-driven simulator for microarchitecture research that is used for data prefetching championships [8], [19]. We simulate an Intel Granite Rapids [20] like memory hierarchy with 2MB L2 and 3MB L3 per core. We simulate 200M instructions after a warmup of 50M instructions. We use workloads from CloudSuite, CVP1, XSBench, Google, and Java Renaissance [13].

Single-core performance. Figure 3 shows the performance achieved with different state-of-the-art prefetchers and their corresponding storage budgets. Toppy, with an offset accuracy threshold of 0.1, outperforms all the prefetchers as Toppy provides an average performance of 9.22% with a storage overhead of less than 1KB. Bingo is the next best prefetcher with an improvement of 6.28%, but with a storage overhead of more than 100KB. Interestingly, Toppy outperforms the best combination of prefetching techniques, Berti, SPP, and Bingo together. The combination provides an improvement of just above 8%, whereas Toppy provides 9.22%. As a co-prefetcher with BOP, Bingo, and IPCP, Toppy provides additional performance improvements, pushing the performance improvement to 11%. On average, Toppy covers 41.48% of the LLC misses.

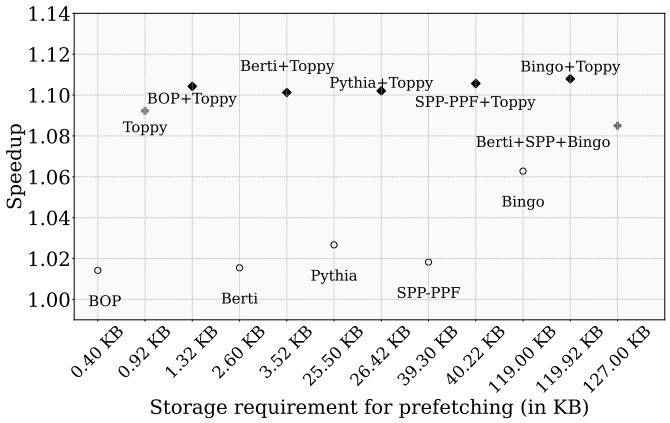

Fig. 3. Toppy and other state-of-the-art prefetchers, including the best combination of prefetchers (Berti+SPP+Bingo) normalized to IP-stride.

TABLE II
TOPPY'S PREFETCH ACCURACY, SPEEDUP, AND DRAM TRAFFIC BASED ON OFFSET ACCURACY. DRAM TPKI OF THE BASELINE IS 12.2.

Offset Accuracy Threshold	Prefetch Accuracy	Performance	DRAM TPKI
0.1	34.84%	9.22%	19.54
0.2	38.57%	8.34%	17.49
0.3	43.96%	7.32%	15.91
0.4	46.76%	6.07%	14.65
0.5	49.57%	4.33%	13.63

Client Workloads from `CVP1` see the maximum benefit in terms of LLC coverage, with an LLC coverage of 70%. The average value of K selected by Toppy for prefetching into LLC is five, which makes it a degree-five prefetcher.

DRAM traffic and prefetch accuracy. Table II shows the prefetch accuracy and DRAM traffic per kilo instructions (TPKI), and speedup based on the offset accuracy thresholds from 0.1 to 0.5. For single core with ample DRAM bandwidth, Toppy uses 0.1 as the offset accuracy threshold, providing coverage at the cost of accuracy, which is low (34.84%). Note that all prefetchers deliver an accuracy of less than 40% except Bingo. As we increase the offset accuracy threshold, Toppy provides better accuracy close to 50% with a threshold of 0.5.

Performance vs DRAM traffic. We evaluate Bingo and Toppy with low DRAM bandwidths, with DRAM bandwidths varying from 400MTPS to 6400 MTPS, with speedup ranging from 3.36% to 9.22%. Toppy outperforms Bingo across different DRAM MTPS values. For 200MTPS, both Bingo and Toppy perform below the baseline.

Multicore performance. Figure 4 shows the multicore performance in the form of normalized weighted speedup. We create 70 4-core heterogeneous mixes by mixing single-core workloads randomly. All four cores share one DRAM channel. Note that even though Toppy is an LLC prefetcher, it still uses the core ID to isolate the global access stream, making it a per-core global offset prefetcher. In terms of performance, Bingo is the best prefetcher, followed by Toppy. The effectiveness of Toppy drops a bit compared to single-core performance, as Toppy does not prefetch aggressively and is conscious of the DRAM bandwidth. LLC pollution affects marginally, as most of the LLC lines are dead on arrival with a 3MB/core LLC.

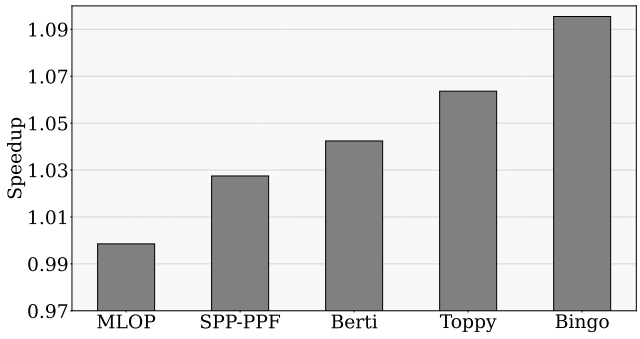

Fig. 4. Multi-core performance (weighted speedup) averaged across 70 mixes.

V. CONCLUSION

We presented Toppy, an LLC-based offset data prefetcher targeted for server workloads. Toppy exploits the fact that server workloads exhibit persistent offsets that persist throughout the program. Toppy uses the Misra-Gries algorithm to find out the persistent offsets at the LLC. We also proposed an adaptive version of Toppy that balances prefetch coverage, accuracy, and DRAM traffic. With a per-core storage overhead of 274 bytes, Toppy provides a lightweight and yet high-performing prefetching option for server workloads.

REFERENCES

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

[2] J. K. et al., "Path confidence based lookahead prefetching," in *49th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

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

[4] M. B. et al., "Bingo spatial data prefetcher," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[5] A. N. torres et al., "Berti: an accurate local-delta data prefetcher," in *55th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2022, pp. 975–991.

[6] Z. et al., " Gaze into the Pattern: Characterizing Spatial Patterns with Internal Temporal Correlations for Hardware Prefetching ," in *HPCA*, Mar. 2025, pp. 173–187.

[7] M. B. et al., "Evaluation of hardware data prefetchers on server processors," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 52:1–52:29, 2019.

[8] "The 4th data prefetching championship (dpc-4)," Feb. 2026. [Online]. Available: https://sites.google.com/view/dpc4-2026/home?authuser=0

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

[10] "The 1st Championship Value Prediction (CVP-1)," https://www.microarch.org/cvp1/cvp1/index.htm, Jun. 2018.

[11] "Xsbench traces for champsim," œXSBench,https://github.com/ANL-CESAR/XSBench, Oct. 2021.

[12] "Google datacenter traces v2 for champsim," https://drive.google.com/drive/folders/1Sh6CE7bfZQgVYZ5lSUwmWggR2wGcFyjJ, Nov. 2024.

[13] A. et al., "Renaissance: benchmarking suite for parallel applications on the jvm," ser. PLDI 2019, p. 31–47.

[14] J. K. et al., "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," p. 737–749, 2017.

[15] M. S. et al., "Multi-lookahead offset prefetching," in *The 3rd Data Prefetching Championship*, Jun. 2019.

[16] A. D. Metwally, A. and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," 2005, pp. 398–412.

[17] R. B. et al., "Dspatch: Dual spatial pattern prefetcher," in *52nd Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2019, pp. 531–544.

[18] "ChampSim simulator," http://github.com/ChampSim/ChampSim, May 2020.

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

[20] "Intel granite rapids," https://en.wikipedia.org/wiki/Granite_Rapids, 2024.