Hardware Data Prefetching for Server Workloads

Submitted in partial fulfillment of the requirements for the degree of
Master of Science (by Research)
by

Naman Sharma
23M2109
namansharma@cse.iitb.ac.in

Supervisor:
Prof. Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2023-2026

Thesis Approval

This thesis entitled

Hardware Data Prefetching for Server Workloads

by

Naman Sharma
Roll No.: 23M2109

is approved for the degree of
Master of Science by Research in Computer Science and Engineering

Digital Signature
Biswabandan Panda (10001949)
29-Jun-26 03:20:32 PM

Prof. Biswabandan Panda
(Supervisor)

Digital Signature
Manas Thakur (10002039)
29-Jun-26 03:53:23 PM

Prof. Manas Thakur
(Examiner)

Devashree Tripathy

Prof. Devashree Tripathy
(External Examiner)

Digital Signature
Udayan Ganguly Ganguly (i10040)
30-Jun-26 05:01:45 PM

Prof. Udayan Ganguly
(Chairperson)

Date: June 28, 2026
Place: IIT Bombay

Declaration

I declare that this written submission represents my ideas in my own words and that wherever the ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented, fabricated, or falsified any idea, data, fact, or source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and may also invoke penal action from the sources that have not been properly cited or from whom proper permission has not been obtained when required.

I further declare that I have used AI-assisted tools, including ChatGPT (OpenAI, GPT-5.5), Grammarly, and Overleaf (LaTeX editor), for language refinement, restructuring of text, formatting assistance, and improving the clarity and coherence of the writing. All technical ideas, analyses, experimental results, observations, and conclusions presented in this thesis are my own work. The AI-assisted tools were not used to generate or fabricate experimental results, data, analyses, or references.

Naman Sharma
23M2109

Date: June 28, 2026

Abstract

Hardware data prefetching is a fundamental technique for mitigating the long latency of off-chip memory accesses and is critical for sustaining the performance of high-performance processors. While prior research has significantly advanced prefetching techniques, most of the techniques are optimized for workloads with relatively small instruction footprints. In contrast, server workloads exhibit very large instruction footprints (tens of megabytes) along with complex and irregular memory access patterns due to frequent transitions between user and kernel execution. These characteristics limit the effectiveness of state-of-the-art hardware prefetchers, particularly those that rely on program counter (PC)-based correlation, as they require substantial metadata storage and struggle to maintain high accuracy and coverage. In this thesis, we observe that server workloads often exhibit a set of persistent global memory access offsets that remain stable across execution. Motivated by this observation, we propose Toppy, a lightweight LLC-based hardware prefetcher that leverages global offset patterns without relying on PC-based tracking. Toppy employs the Misra–Gries heavy-hitter algorithm to efficiently identify the top-k most frequent offsets using a very small hardware footprint. These persistent offsets are then used to generate prefetch requests at the last-level cache, enabling effective capture of recurring access patterns with minimal overhead. To further improve robustness, we design an adaptive mechanism that dynamically balances prefetch coverage, accuracy, and DRAM traffic by tuning the offset selection threshold at runtime. This allows Toppy to achieve a favorable trade-off between performance and memory bandwidth consumption across different workload characteristics and system configurations. We evaluate Toppy on a diverse set of server and cloud workloads and compare it against several state-of-the-art prefetchers, including KPC-P, IPCP, Berti, SPP-PPF, Gaze, Bingo, Pythia, and MLOP. Compared to a baseline system with an L1D IP-stride prefetcher, Toppy achieves an average performance improvement of 9.22%, outperforming all evaluated designs. The next best prefetcher, Bingo, achieves a performance gain of 6.28%. Importantly, Toppy delivers this performance with a storage overhead of only 425 bytes per core, compared to approximately 119 KB required by Bingo. Overall, Toppy demonstrates that exploiting persistent global offsets is an effective approach for prefetching in server workloads. It achieves a strong balance between performance improvement, hardware cost, and memory bandwidth utilization, making it a practical solution for future server-class processors.

Contents

Chapter 1

Introduction

The growing disparity between processor speeds and main memory latency has made memory access one of the primary performance bottlenecks in modern computing systems. This challenge is particularly pronounced in large-scale server workloads, where applications operate on massive datasets and exhibit complex memory access behaviors. Hardware data prefetching has long been recognized as an effective technique to mitigate memory latency by predicting future memory accesses and fetching data into the cache hierarchy ahead of demand [35, 33, 40].

A hardware prefetcher operates alongside the cache subsystem by observing memory access patterns at runtime and issuing speculative memory requests. Over the years, a wide variety of prefetching techniques have been proposed, ranging from simple stride-based mechanisms to sophisticated correlation-based and learning-based approaches [13, 38, 18]. While these techniques have demonstrated significant performance improvements on traditional benchmark suites such as SPEC CPU [5], their effectiveness on modern server workloads remains limited [14].

Recent studies have shown that state-of-the-art hardware prefetchers provide only marginal performance improvements for memory-intensive server applications [14], as illustrated in Figure 1.1. In many cases, the observed gains are less than 2% compared to a baseline IP-stride prefetcher at L1D. Even advanced designs, including temporal prefetchers such as ISB [47], Triage [46], Triangel [10], and reinforcement learning-based approaches such as Pythia [15], fail to consistently deliver substantial benefits. Among existing techniques, the Bingo prefetcher achieves the highest improvement of approximately 6.28% [13], but at the cost of significant storage overhead. This highlights a fundamental gap between prefetcher design assumptions and the characteristics of modern server workloads.

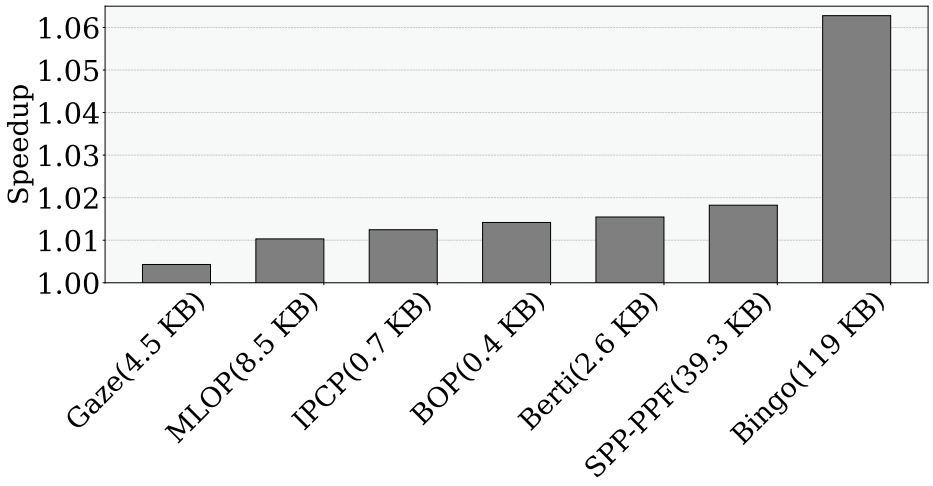


Figure 1.1: Performance of hardware prefetchers for memory-intensive server workloads normalized to an L1D IP-stride prefetcher. Most prefetchers provide less than 2% improvement, while Bingo achieves the highest improvement of 6.28%.

One of the primary reasons for this inefficiency is the unique nature of server applications. Unlike traditional workloads, server workloads exhibit highly irregular and diverse memory access patterns, resulting in significantly lower prefetch accuracy, as shown in Figure 1.2. Prefetchers that achieve high accuracy on conventional benchmarks often experience a drastic drop in effectiveness when applied to server environments [14]. Furthermore, temporal prefetchers rely on recurring access patterns, which are often absent in these workloads, further limiting their utility [47, 46, 10].

Another critical challenge is the rapidly increasing instruction footprint of server applications. Studies indicate that instruction footprints grow by approximately 30% annually [11]. This leads to a substantial increase in the number of unique program counters (PCs) encountered during execution. As a result, PC-based prefetchers face significant scalability issues, as they require large metadata tables to track per-PC behavior. As shown in Figure 1.3, increasing the storage capacity of the Bingo prefetcher results in only marginal performance gains [13], highlighting the limited scalability of large metadata-based designs.

Non-PC-based prefetchers attempt to address these scalability concerns by eliminating dependence on program counters [34]. However, these approaches typically rely on detecting short-term or transient access patterns, which are often irregular in server workloads. Consequently, their performance improvements remain limited and comparable to PC-based designs [14].

In addition to these challenges, cache hierarchy interactions further complicate prefetch-

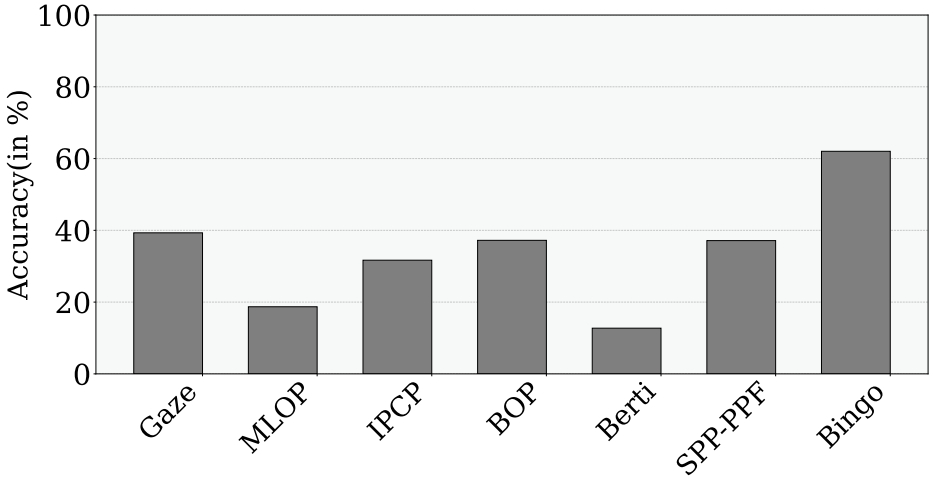


Figure 1.2: Prefetch accuracy of state-of-the-art prefetchers averaged across memory-intensive server workloads. Most prefetchers exhibit low accuracy, indicating irregular memory access patterns.

ing in server systems. High instruction miss rates at the L1 and L2 caches lead to contention between instruction and data streams. Data prefetching at lower cache levels can introduce cache pollution, evicting useful instruction lines and causing front-end bottlenecks [37, 30]. This behavior often negates the benefits of accurate data prefetching, necessitating careful consideration of where prefetching should be performed in the memory hierarchy.

These observations motivate the need for a fundamentally different approach to data prefetching for server workloads. An effective solution must be lightweight and capable of handling irregular access patterns without relying on program counters. Additionally, it should minimize interference with instruction caches and operate efficiently within constrained hardware resources. In this work, we observe that despite the apparent irregularity of server workloads, a small number of global memory access offsets account for a significant fraction of cache accesses. These offsets, defined as the differences between consecutive cache-line addresses, exhibit persistence across execution phases. We find that a small set of frequently occurring global offsets can cover a substantial portion of last-level cache accesses, providing an opportunity for efficient prefetching.

Motivated by this insight, we propose *Toppy*, a lightweight offset-based hardware prefetcher that leverages the most frequent global offsets to generate prefetch requests. Unlike prior approaches [35], Toppy does not rely on program counters or localized access streams. Instead, it dynamically identifies persistent offsets using a streaming algorithm and issues prefetches based on their observed frequency. By operating at the last-level cache and adapting its

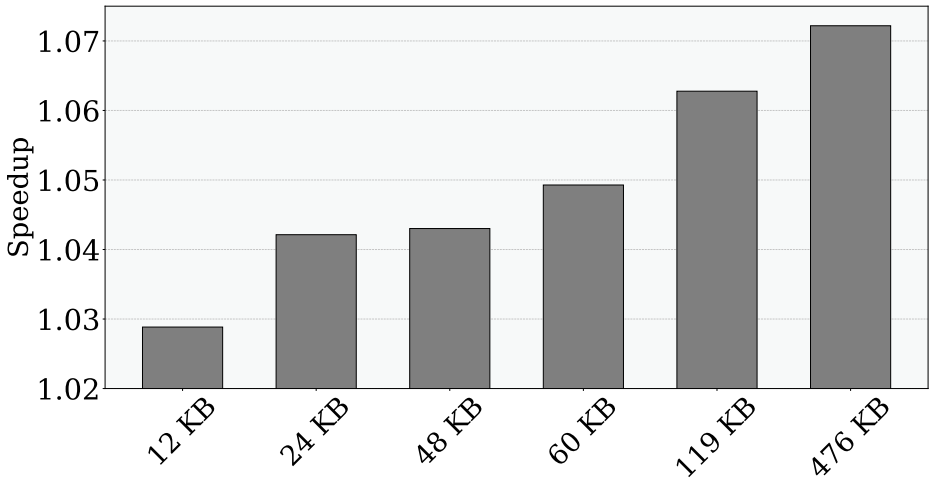


Figure 1.3: Impact of increasing storage size on Bingo prefetcher performance. Even huge increases in storage result in only marginal performance gains.

aggressiveness based on runtime conditions, Toppy achieves a balance between performance improvement and storage overhead. Our evaluation demonstrates that Toppy significantly outperforms existing state-of-the-art prefetchers on memory-intensive server workloads while requiring only a tiny storage overhead. These results highlight the effectiveness of leveraging global offset locality and motivate a shift toward simpler, more effective prefetching designs.

Chapter 2

Background & Motivation

This chapter presents the background and motivation necessary to understand the design and challenges addressed in this thesis. We first discuss the organization of the memory hierarchy and the need for multiple levels of memory to balance performance, cost, etc. We then examine the impact of memory access latency on processor performance and describe the *Memory Wall* problem, which arises due to the growing gap between processor speeds and memory access times.

To mitigate memory access latency, modern processors employ several hardware optimizations such as caching, out-of-order execution, and hardware prefetching. Among these, prefetching plays an important role in improving memory-level parallelism by predicting future memory accesses and fetching data before it is requested by the processor. This chapter also reviews the fundamentals of hardware prefetching, discusses prior prefetching techniques, and motivates the need for lightweight and bandwidth-aware prefetching mechanisms for systems.

2.1 Memory Hierarchy

Modern processors employ a hierarchical memory organization consisting of multiple levels of cache positioned between the processor cores and main memory, as shown in Figure 2.1. This hierarchy is designed to bridge the large gap between processor speed and main memory latency by exploiting the trade-off between access latency and capacity.

At the top of the hierarchy is the **L1 data cache (L1D)**, which is closest to the processor core and provides the lowest access latency, typically on the order of a few nanoseconds. Due to its stringent latency requirements, the L1D cache is very small, usually only tens of

kilobytes in size. Its primary role is to supply data to the core at nearly core speed, thereby minimizing pipeline stalls.

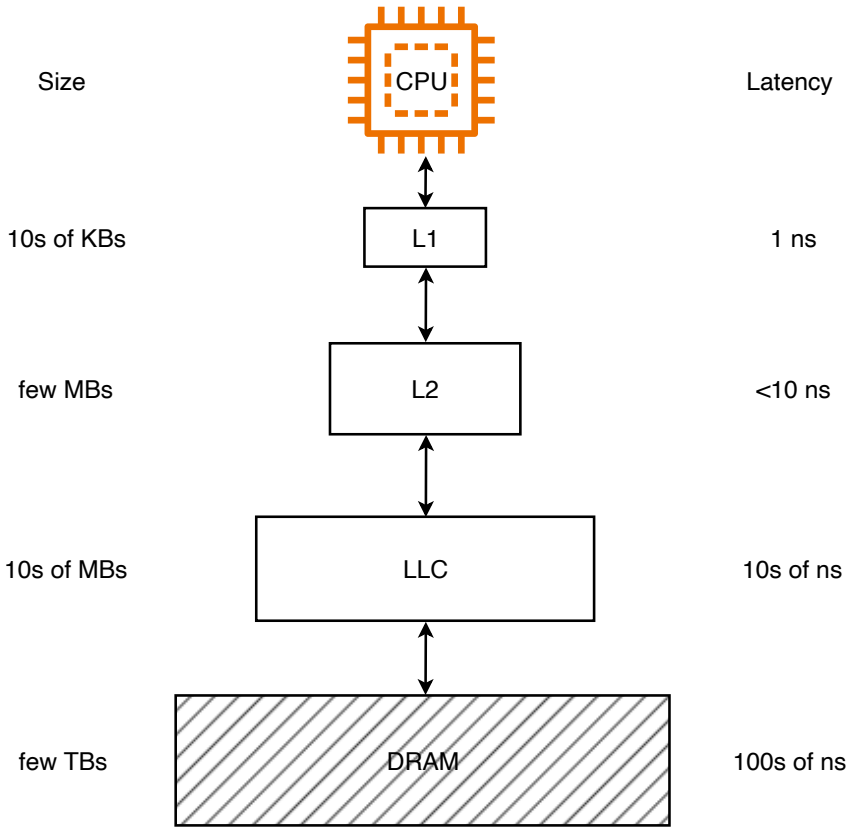


Figure 2.1: Overview of a memory hierarchy

Below the L1D cache lies the **Level-2 cache (L2C)**. The L2 cache is larger than the L1 cache, typically on the order of hundreds to thousands of kilobytes, and has slightly higher access latency. It serves as an intermediate cache that captures working sets too large for the L1 cache while still providing much faster access than lower levels of the memory hierarchy. In most modern architectures, both the L1D and L2 caches are **private to each core**, allowing low-latency access.

The **Last-Level Cache (LLC)** is the largest on-chip cache and is commonly shared among all processor cores. With capacities ranging from a few megabytes to tens of megabytes, the LLC is designed to retain a large amount of data on-chip and reduce accesses to main memory. Although its access latency is higher, typically tens of nanoseconds, it remains significantly faster than accessing main memory.

At the bottom of the hierarchy is **main memory (DRAM)**, which offers large capacity, often several gigabytes to terabytes, but at the cost of much higher access latency, typically on the order of hundreds of nanoseconds. Memory accesses that miss in all cache levels must be served by DRAM, resulting in substantial performance penalties.

Overall, the memory hierarchy illustrated in Figure 2.1 demonstrates the fundamental design principle that as memory capacity increases, access latency also increases. This motivates the use of effective caching and hardware prefetching techniques to mitigate memory access delays and improve system performance.

2.2 Server Processors & Server Workloads

2.2.1 Server processors

Server processors, including Intel Xeon [27] and AMD EPYC [9], are designed to sustain high throughput for large-scale data center workloads. These systems integrate a large number of cores, support simultaneous multithreading (SMT), and employ deep memory hierarchies with private L1/L2 caches and a shared last-level cache (LLC) to efficiently manage data movement. Internally, these processors rely on out-of-order execution to extract instruction-level parallelism and maximize resource utilization. While the front-end is responsible for supplying instructions, overall performance is also constrained by the back-end, where instructions are scheduled, executed, and retired.

A key challenge in server processors lies in the memory subsystem. Modern workloads exhibit large working sets and irregular access patterns, which place significant pressure on the cache hierarchy. Frequent data cache and TLB misses result in long-latency memory accesses, causing pipeline stalls and underutilization of execution resources. To mitigate memory latency, processors employ multi-level cache hierarchies along with hardware data prefetchers that attempt to predict future memory accesses. While effective for regular access patterns, these prefetchers often struggle with the irregular and complex behaviors observed in server workloads, leading to limited coverage and reduced accuracy.

Although front-end optimizations such as branch prediction, Fetch Target Queues (FTQ), and instruction prefetching techniques like Fetch-Directed Instruction Prefetching (FDIP) help maintain instruction supply, they do not eliminate performance bottlenecks. Prior work [11] shows that a significant portion of execution time is spent in both front-end and back-end stalls. Figure 2.2 presents a breakdown of pipeline stall cycles on an Intel Xeon

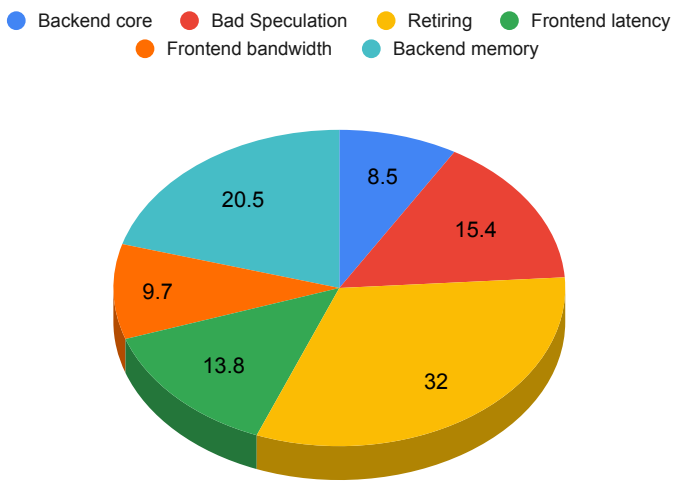


Figure 2.2: CPU performance breakdown in server workloads on an Intel Xeon processor (adapted from [11]).

processor. While front-end stalls contribute notably to performance loss, a substantial fraction of cycles is dominated by back-end stalls, particularly those caused by memory latency, cache misses, and contention for execution resources.

In practice, even with efficient instruction delivery, back-end bottlenecks remain a primary limitation to performance. Data-intensive server applications frequently experience long memory access latencies and irregular access patterns, which reduce effective cache utilization and stall execution pipelines. Therefore, improving back-end efficiency, particularly in the memory subsystem, is critical for enhancing overall system performance. Addressing challenges such as memory latency, cache behavior, and prefetching effectiveness is essential to fully exploit the capabilities of modern server processors.

2.2.2 Server Workloads

A rigorous understanding of server workloads is a prerequisite for any architectural investigation of processor performance. A workload represents the complete set of computational activities required to execute an application, service, or task, encompassing not only the software itself but also the associated demands on compute, memory, storage, and networking resources [25]. Importantly, workloads are characterized not only by their functional role but also by the intensity and diversity of resource usage they impose on the underlying

hardware. As a result, two workloads serving similar purposes may exhibit vastly different architectural requirements depending on their execution characteristics [25].

Server workloads are typically categorized along both functional and behavioral dimensions. From a functional perspective, transactional workloads process a large number of small, latency-sensitive operations, commonly observed in financial systems, e-commerce platforms, and request-driven services where responsiveness is critical [42]. In contrast, analytical workloads operate on large datasets and perform compute-intensive operations such as aggregations, data mining, and machine learning inference, often prioritizing throughput over latency [26]. Batch workloads form another important category, where large volumes of tasks are executed in parallel during off-peak periods, leveraging thread-level and data-level parallelism to maximize system utilization [26].

To capture the diversity of modern server workloads in our study, we utilize publicly available traces from multiple benchmark suites, including CloudSuite [2], CVP-1 [3] traces provided by Qualcomm, Java Renaissance workloads from a recent MICRO 2024 study [44], and Google workloads released as part of the Data Prefetching Championship (DPC-4) [8]. These workloads collectively span a broad range of real-world applications, including cloud services, enterprise systems, and large-scale production environments, and exhibit diverse memory access characteristics. Modern server workloads have grown significantly in complexity due to their interaction with deep and layered software stacks. These stacks include operating system components, kernel services [31, 23], large third-party libraries [12], and managed runtime environments such as virtual machines and language runtimes [39]. This increasing software complexity leads to a rapid expansion in code footprint, with studies indicating an approximate growth rate of 30% per year [31, 12]. Consequently, both instruction and data working sets have expanded significantly, placing substantial pressure on the processor memory hierarchy.

This behavior is quantitatively illustrated in Table 2.1, which presents instruction and data Misses Per Kilo Instructions (MPKI) across multiple cache levels for the evaluated workloads. A key observation is the substantially higher instruction cache miss rate in server workloads compared to traditional benchmarks. The average L1 instruction cache (L1I) MPKI across server workloads is 19.18, which is over 11× higher than SPEC CPU 2017 at 1.63. Certain workloads, such as CVP-1, exhibit extremely high instruction pressure with an L1I MPKI of 32.15, while Google workloads reach 18.17. Even relatively moderate workloads such as CloudSuite and Java Renaissance show L1I MPKI values exceeding 10. This clearly indicates that server workloads have significantly larger instruction footprints,

Table 2.1: Instruction and Data MPKI across cache levels for evaluated workloads

Workload	**L1I MPKI**	**L1D MPKI**	**L2 IMPKI**	**L2 DMPKI**	**LLC IMPKI**	**LLC DMPKI**
CloudSuite	10.77	6.69	0.09	3.54	0.00	2.66
Java Renaissance	11.76	6.02	0.78	3.17	0.05	2.65
XS Bench	0.00	23.00	0.00	59.03	0.00	27.21
Google	18.17	10.79	3.83	4.21	0.81	6.34
CVP-1	32.15	14.80	0.29	45.04	0.00	13.60
Average	**19.18**	**11.93**	**2.85**	**7.34**	**0.34**	**5.52**
SPEC CPU 2017	1.63	30.51	0.19	17.89	0.00	12.79

often exceeding the capacity of the L1I cache, resulting in frequent instruction misses.

In contrast, the data-side behavior exhibits a different trend. SPEC CPU 2017 shows a higher average L1D MPKI of 30.51 compared to 11.93 for server workloads, suggesting that traditional benchmarks are more data-intensive at the L1 level. However, server workloads demonstrate more diverse and irregular behavior across deeper levels of the memory hierarchy. For instance, XS Bench shows a high L1D MPKI of 23.00 along with an elevated LLC MPKI of 27.21, indicating sustained pressure on lower cache levels. Google workloads exhibit relatively balanced pressure across levels, with L2 MPKI values and LLC MPKI of 6.34. On the other hand, CVP-1 shows moderate LLC MPKI of 13.60.

Overall, while the average LLC MPKI of server workloads (5.52) is lower than that of SPEC CPU 2017 (12.79), this does not imply reduced memory pressure. Instead, it reflects a fundamental shift in workload characteristics: server workloads are increasingly dominated by large instruction footprints and irregular access patterns, rather than purely data-intensive behavior.

2.3 Hardware Data Prefetchers

Prefetching algorithms play a vital role in improving computer system performance by predicting future memory accesses. As illustrated in Figure 2.3, (❶) the processor core issues demand requests to the cache following a sequential access pattern, such as X, X + 1, and X + 2. (❷) The hardware prefetcher monitors these cache accesses and identifies the underlying sequential pattern. Based on this observation, it predicts the next memory address, X + 3, and generates a prefetch request for it. (❸) The corresponding data is then fetched and brought into the cache ahead of time. (❹) Consequently, when the processor later issues

a demand request for X + 3, the data is already available in the cache, thereby reducing memory access latency and avoiding the delay associated with accessing main memory.

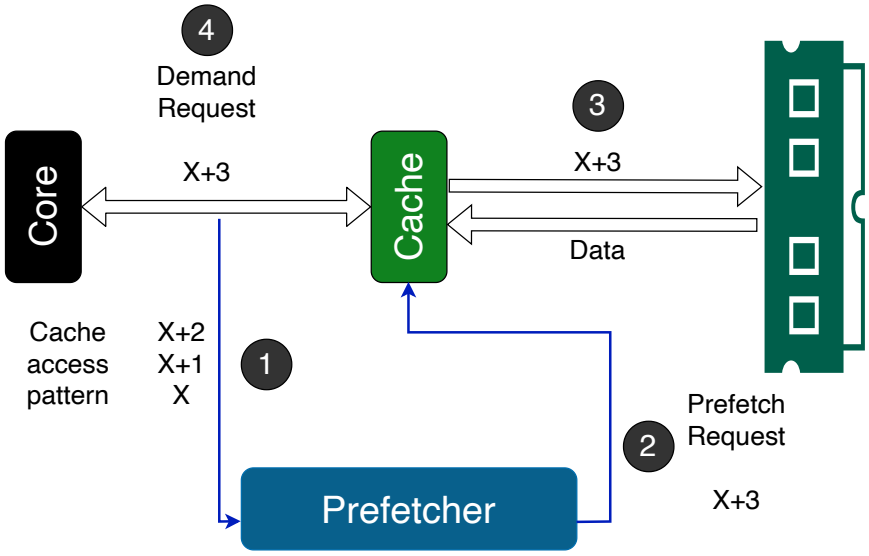


Figure 2.3: Example of hardware prefetching based on memory access patterns.

Prefetching enhances computational efficiency by minimizing memory latency and reducing the processor's idle time. Prefetchers, in combination with caches, are crucial for bridging the gap between processor speeds and memory access times, enabling seamless data flow and improving the overall performance of modern computer systems.

Several key metrics are used to evaluate prefetcher performance. **Accuracy** refers to the fraction of prefetched cache lines that are actually used by the processor before eviction. High accuracy indicates that the prefetcher is making correct predictions, while low accuracy implies wasted bandwidth and cache pollution. In this context, **useful prefetches** are those prefetched cache lines that are later accessed by the processor and contribute to demand requests. In contrast, **useless prefetches** are those that are never accessed by the program and may displace useful data from the cache, negatively impacting performance.

Another critical metric is **coverage**, defined as the fraction of total cache misses that are eliminated due to prefetching. A high-coverage prefetcher successfully brings a large portion of missing data into the cache ahead of time, thereby reducing demand misses. However, high coverage alone is not sufficient; it must be balanced with accuracy to ensure that the prefetched data is actually beneficial.

Another important design parameter is the **prefetch degree**, which specifies the number

of cache lines a prefetcher requests ahead of a detected access pattern. For example, if a sequential prefetcher observes accesses to cache lines X, X+1, and X+2, a degree of one would prefetch only X+3, whereas a degree of four would prefetch X+3 through X+6. A higher prefetch degree can improve coverage by bringing more future data into the cache, increasing the likelihood that upcoming memory accesses will find their data already available. However, excessively large prefetch degrees may reduce accuracy, consume additional memory bandwidth, and increase cache pollution by introducing data that is never used. Therefore, selecting an appropriate prefetch degree is critical for balancing performance benefits against resource overheads.

Timeliness captures whether prefetched data arrives in the cache at the right time. A prefetch is considered timely if it arrives before the corresponding demand access but not so early that it is evicted before use. **Late prefetches**, which arrive after the demand request, fail to hide memory latency, while early prefetches risk being evicted or causing unnecessary cache pressure. Achieving good timeliness is particularly challenging in irregular workloads where access patterns are difficult to predict.

Prefetching also introduces trade-offs in terms of bandwidth consumption and cache utilization. Aggressive prefetching strategies may increase coverage but can overwhelm memory bandwidth and increase cache contention. These fundamental concepts provide the basis for understanding modern prefetching designs. Building on this foundation, the next subsections review state-of-the-art prefetching techniques proposed in prior work.

2.3.1 BOP: Best-Offset Hardware Prefetcher

The Best-Offset Prefetcher (BOP) [35] is a hardware prefetching mechanism designed to identify a single memory access offset that maximizes the number of *timely* prefetches. A prefetch is considered *timely* when the prefetched cache line arrives in the cache hierarchy before it is requested by the processor, thereby successfully hiding memory access latency. In contrast, a *late prefetch* corresponds to a correct prediction issued too close to the demand access, resulting in pipeline stalls and reduced effectiveness.

BOP focuses on identifying the best offset for generating prefetch requests. An offset is defined as the difference between two cache line addresses. For example, consider three accesses to cache line addresses x, $x+5$, and $x+7$. The possible offsets are $(x+5-x)=5$, $(x+7-(x+5))=2$, and $(x+7-x)=7$. BOP gained prominence as the winning design of the **Second Data Prefetching Championship (DPC-2)** [1], demonstrating strong

performance due to its emphasis on timeliness and simplicity. The core idea behind BOP is to dynamically evaluate a set of candidate offsets and select the one that yields the highest number of useful prefetches over a fixed observation window, referred to as an *epoch*.

The prefetcher maintains a predefined list of candidate offsets, determined at design time, which represent possible spatial distances between memory accesses. During each epoch, BOP evaluates these offsets by issuing trial prefetches (or by monitoring hypothetical usefulness) and tracking their effectiveness using lightweight hardware structures. Specifically, for each candidate offset, the prefetcher records the number of times the corresponding prefetched cache line is accessed before eviction, thereby classifying prefetches as either useful (timely) or ineffective (late or unused).

At the end of each epoch, BOP selects the offset that produces the highest number of timely prefetch hits as the *best offset*. This selected offset is then used exclusively in the subsequent epoch to generate prefetch requests. By restricting prefetching to a single offset, BOP avoids unnecessary memory traffic and reduces cache pollution, leading to high prefetch accuracy and efficient bandwidth utilization.

However, this design choice also introduces a key limitation. Since BOP relies on a single global offset at any given time, it is inherently constrained in its ability to capture complex or irregular memory access patterns that may require multiple offsets or context-dependent behavior. As a result, while BOP excels in scenarios with regular and dominant spatial patterns, its *coverage* is limited in workloads exhibiting diverse or multi-modal access characteristics. Overall, BOP represents a design point that prioritizes timeliness and accuracy through selective prefetching, trading off coverage and adaptability for simplicity and robustness.

2.3.2 MLOP: Multi-Lookahead Offset Prefetcher

The Multi-Lookahead Offset Prefetcher (MLOP) [45] extends the design philosophy of the Best-Offset Prefetcher (BOP) by addressing its fundamental limitation of relying on a single offset and issuing only one prefetch per demand access. While BOP focuses on selecting a single globally optimal offset that maximizes timeliness, MLOP generalizes this approach by learning and applying *multiple offsets* corresponding to different *lookahead levels*, thereby increasing both prefetch degree and coverage.

At its core, MLOP is designed to answer the following key question: *for a given cache miss, which combination of offset and lookahead distance can generate a timely prefetch?* To

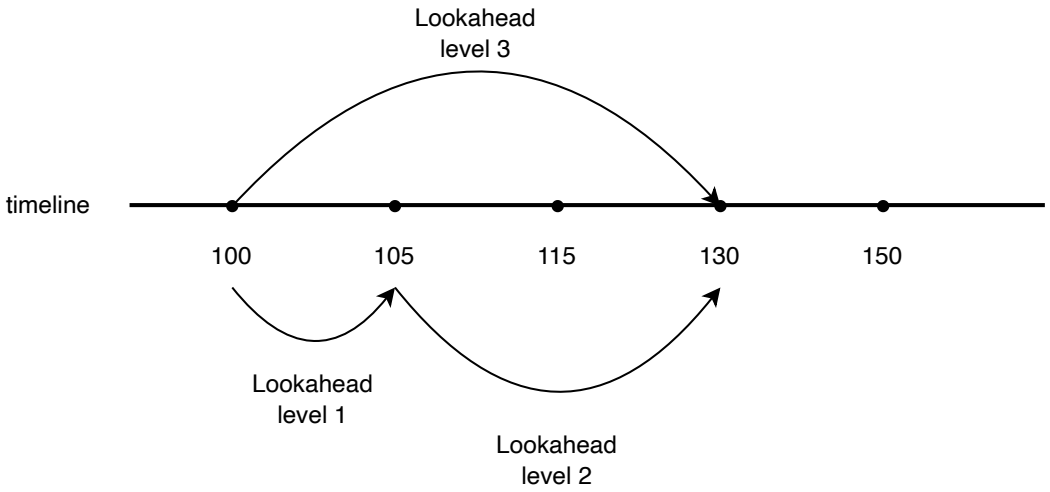


Figure 2.4: Example of lookahead levels applied to a sequence of cache line addresses.

this end, MLOP evaluates a set of candidate offsets across multiple lookahead levels, where each lookahead level represents a different distance into the future access stream. Instead of associating a single offset with a demand access, MLOP allows multiple offsets to be active simultaneously, each responsible for prefetching data at a different future point.

Lookahead in MLOP refers to the relative distance between the current demand access and the predicted future access that a prefetch aims to target. A lookahead of one corresponds to the immediate next access, while higher lookahead levels correspond to accesses further ahead in the stream. This notion enables MLOP to generate prefetches with varying lead times, which is critical for hiding memory latency, especially in deep memory hierarchies.

As illustrated in Figure 2.4, consider a demand access to cache line address 100. A prefetch to address 105 corresponds to lookahead level 1, as it targets the immediate next access. Similarly, prefetches to addresses 115 and 130 correspond to lookahead levels 2 and 3, respectively, since they target accesses that occur further ahead in the sequence. By enabling such multi-level predictions, MLOP can issue prefetches early enough to overlap memory latency for accesses that are distant in time.

This multi-offset, multi-lookahead design allows MLOP to issue multiple prefetch requests per demand access, significantly increasing the prefetch degree. As a result, MLOP achieves higher coverage compared to BOP, particularly for workloads with complex memory access

patterns that cannot be captured by a single offset. Furthermore, by associating different offsets with different lookahead levels, MLOP improves prefetch timeliness, ensuring that data required in both the near and distant future is fetched appropriately.

2.3.3 Berti: an Accurate Local-Delta Data Prefetcher

Berti [38] is an L1D prefetcher that leverages a local delta–based mechanism to proactively fetch data. It distinguishes between *stride* and *delta*: a stride is defined as the difference between two consecutive memory accesses generated by the same instruction pointer (IP), whereas a delta represents the difference between the current memory access and any previous memory access generated by the same IP. By tracking and learning useful deltas through several hardware structures, Berti improves its ability to anticipate future memory accesses and issue timely prefetch requests.

In this context, a local delta refers to the difference between the cache-line address of the current memory access and that of a previous access generated by the same instruction pointer (IP). Unlike a stride, which considers only two consecutive accesses, a local delta can be computed with respect to any prior access from the same IP. This allows Berti to capture a broader range of access patterns beyond simple sequential behavior.

For example, consider an instruction IP_X that generates accesses to cache line addresses X, $X+2$, $X+5$, $X+7$, and $X+10$. The differences between consecutive accesses are +2, +3, +2, and +3, resulting in a repeating but non-constant access pattern. A stride predictor, which relies on a single constant stride, may not accurately capture such behavior. When the access to $X+10$ occurs, the set of observable local deltas includes +10 (computed as $X+10-X$), +8 (from $X+10-(X+2)$), +5 (from $X+10-(X+5)$), and +3 (from $X+10-(X+7)$). Unlike stride-based prediction, which considers only the most recent difference, local deltas capture relationships with multiple prior accesses.

This distinction between strides and local deltas is illustrated in Figure 2.5, adapted and modified from [38]. By maintaining such a set of deltas, Berti is able to recognize multiple potential deltas that may be useful for future predictions. To understand the importance of timeliness, consider a scenario where the goal is to prefetch cache line address 15. After observing demand accesses to addresses 11 and 8, the prefetcher can compute deltas of +4 and +7, respectively. While these deltas correctly identify the target address, issuing prefetches based on them may not effectively hide L1D miss latency, since the prefetch requests are generated too close to the actual demand access, resulting in late prefetches.

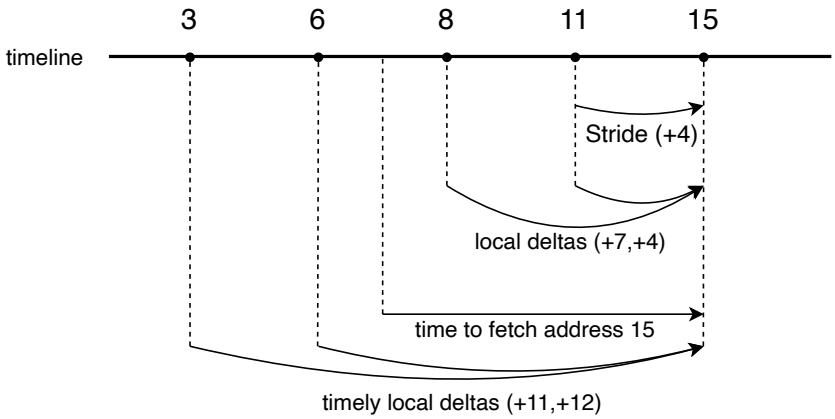


Figure 2.5: Illustration of how the Berti prefetcher derives stride, local delta, and timely local delta information from memory access sequences to enable timely prefetching.

As shown in Figure 2.5, accesses to addresses 6 or 3 can generate larger deltas of +11 and +12, respectively. Leveraging such deltas enables the prefetcher to initiate requests for address 15 significantly earlier in the access stream. This increased lead time allows the memory subsystem to service the prefetch before the demand access occurs, thereby improving prefetch timeliness and increasing the likelihood of latency hiding.

Overall, Berti's use of local deltas, particularly those derived from earlier accesses, allows it to overcome the limitations of traditional stride-based prefetchers. By identifying both accurate and timely correlations, it strikes a balance between correctness and early prediction, which is critical for achieving high prefetch effectiveness in modern out-of-order processors.

2.3.4 IPCP: Instruction Pointer Classifier-Based Spatial Hardware Prefetching

The IPCP [40] proposes an instruction pointer (IP) classifier-based approach to spatial hardware prefetching, motivated by the observation that different IPs exhibit distinct spatial memory access behaviors depending on the application's control-flow and data-flow characteristics. By dynamically classifying IPs according to their access patterns, the framework enables the use of specialized and lightweight prefetching mechanisms tailored to each class, thereby improving both prefetch accuracy and timeliness.

The Instruction Pointer Classification-based Prefetching (IPCP) [40] framework performs IP classification at the L1 data cache (L1-D) level and uses this information to guide effi-

cient L1-D prefetching. This classification information is further communicated to the L2 prefetcher, enabling coordinated multi-level prefetching with minimal additional hardware overhead. IPCP is designed to handle a wide range of access behaviors.

IPCP categorizes instruction pointers (IPs) based on their observed spatial memory access patterns and applies different prefetching strategies accordingly. IPs exhibiting constant strides between consecutive cache-line accesses are handled using a simple stride prefetcher, while those with irregular or repeating stride patterns are managed using a more complex stride predictor that leverages hashed signatures to capture recent access history and predict future strides with higher confidence. Beyond IP-based patterns, IPCP also identifies global stream behavior, where accesses occur in bursts within a confined memory region, and applies region-based prefetching to preserve spatial locality across multiple IPs. For accesses that do not match these patterns, IPCP cautiously employs next-line prefetching under low memory-pressure conditions to avoid performance degradation. Overall, IPCP combines multiple lightweight and specialized prefetching techniques with adaptive control, enabling effective coverage of diverse access patterns while maintaining low hardware overhead.

2.3.5 Signature Path Prefetcher

Traditional prefetchers, such as stride-based designs, are effective when memory accesses follow simple and regular patterns. However, many modern applications exhibit complex access sequences that cannot be accurately captured using a single stride or offset. As a result, conventional prefetchers often fail to achieve high coverage for these workloads.

The Signature Path Prefetcher (SPP) [32] addresses this limitation by learning correlations between sequences of memory accesses rather than relying solely on individual strides. Instead of predicting future accesses based on a single recent access, SPP captures the history of observed access patterns and uses it to identify recurring memory access paths. This enables the prefetcher to recognize complex patterns that may reappear over time, even when individual offsets appear irregular.

A key feature of SPP is its use of confidence-based prediction. Multiple candidate future accesses may be associated with a given access history, and SPP assigns a confidence value to each prediction based on its past occurrence frequency. These confidence values allow the prefetcher to prioritize more reliable predictions while reducing the impact of inaccurate ones.

By combining path-based correlation with confidence estimation, SPP is able to achieve

higher coverage than traditional stride-based prefetchers and can effectively capture a wider range of memory access behaviors. Consequently, SPP has become a representative design for learning complex memory access patterns in modern processors.

2.3.6 Bingo

Many memory access patterns cannot be accurately captured using a single correlation source. Some accesses are strongly correlated with detailed context, while others can only be predicted using more general information. Relying on a single predictor often leads to either low coverage or low accuracy.

Bingo [13] addresses this challenge by employing multiple correlation mechanisms with varying levels of specificity. Instead of using a single access history, Bingo attempts to match memory accesses using both detailed and coarse-grained context information. When highly specific correlations are available, Bingo generates accurate predictions. When such correlations are unavailable, it falls back to more general correlations to maintain coverage.

This hierarchical prediction strategy enables Bingo to balance prediction accuracy and coverage more effectively than traditional prefetchers. By exploiting correlations at multiple levels of granularity, Bingo can capture a wider range of memory access behaviors while avoiding excessive reliance on any single prediction source.

As a result, Bingo achieves high coverage and strong performance across diverse workloads, making it one of the most effective correlation-based prefetchers proposed in recent years.

2.3.7 Gaze

Modern applications often access memory in ways that are difficult for traditional prefetchers to anticipate. Techniques such as stride and correlation prefetching work well when memory accesses follow clear, repetitive patterns, but many real-world workloads exhibit behavior that appears irregular and unpredictable. However, this apparent randomness is often a consequence of looking too closely at individual memory accesses. When memory activity is viewed at a larger scale, meaningful patterns frequently emerge.

GAZE [19] is based on the insight that applications tend to spend most of their time accessing a relatively small number of active memory regions, which they call zones. Instead of focusing on individual addresses, GAZE tracks these zones and observes how applications move between them over time. By learning the relationships among frequently accessed

zones, GAZE can uncover higher-level structure in memory behavior and use it to predict future accesses more effectively, even when fine-grained access patterns seem irregular.

2.3.8 Chimera: Leveraging Hybrid Offsets for Efficient Data Prefetching

Chimera [24] is a hybrid prefetcher that improves prefetching effectiveness by leveraging multiple memory access features rather than relying on a single dimension such as temporal, spatial, or instruction-based locality. It combines these complementary features to better capture the diverse access patterns commonly observed in modern workloads.

Specifically, Chimera integrates three coordinated components. First, it employs a program counter (PC)-based prefetcher that learns access behavior per instruction and identifies the most useful offsets based on historical patterns, similar in spirit to prior PC-based approaches such as Berti [38]. Second, it incorporates a spatial or region-based prefetcher that analyzes access patterns within a memory region and determines effective offsets using region-level history, similar to techniques used in prefetchers such as Bingo [13] and SPP [33]. Finally, Chimera includes a global or periodic prefetcher that captures recurring access patterns across the entire application by learning global offsets over time, similar to BOP [35].

By combining PC-based, spatial, and global perspectives, Chimera is able to improve both coverage and accuracy while maintaining timeliness. This multi-dimensional design allows it to adapt to a wide range of access behaviors, making it particularly effective for complex and irregular workloads where single-feature prefetchers often fall short.

2.3.9 Hyperion: A Highly Effective Page and PC Based Delta Prefetcher

Hyperion [20] is a hardware prefetcher that extends the timely delta concept introduced in prior work, such as Berti, by incorporating additional contextual information through page-level learning. A timely delta refers to an address difference that enables prefetch requests to arrive at the cache at the right time—neither too early, which may lead to eviction, nor too late, which fails to hide memory latency.

Unlike traditional PC-based approaches that rely solely on the instruction pointer to learn useful deltas, Hyperion augments this learning process by incorporating page-level

information. By considering both the generating instruction and the memory region (page) in which accesses occur, Hyperion captures richer context about memory behavior. This allows it to better distinguish between access patterns that may appear similar at the instruction level but differ across memory regions.

Through this combined use of PC-based and page-based learning, Hyperion improves the accuracy and timeliness of its prefetch decisions. This design enables it to adapt more effectively to complex and irregular workloads, where relying on a single source of information is often insufficient.

2.3.10 Pythia

Pythia [16] is a reinforcement learning (RL)-based hardware prefetcher that focuses on improving overall system performance rather than relying solely on traditional metrics such as coverage and accuracy. It recognizes that these metrics do not always correlate with performance gains and therefore incorporates system-level factors, such as memory bandwidth utilization, into its decision-making process.

Pythia models prefetching as a learning problem, where it selects actions based on runtime behavior and receives a numerical reward that reflects the effectiveness of each prefetch. This reward captures aspects such as accuracy, timeliness, and overall system impact, allowing the prefetcher to adapt dynamically to different workloads.

To capture diverse access patterns, Pythia leverages both control-flow and data-flow information. The control-flow component uses instruction-level context, while the data-flow component captures relationships between memory accesses, enabling more accurate and timely prefetching.

2.4 Motivation

Although hardware data prefetching has been extensively studied, its effectiveness on modern server workloads remains limited. Existing prefetchers provide only modest performance gains due to irregular memory access patterns and large instruction footprints. This section highlights the key limitations and insights that motivate our design.

2.4.1 Irregular Local Offsets

Many prefetchers rely on program counter (PC)-based correlation, assuming that memory accesses from a given PC follow predictable patterns such as fixed strides. However, this assumption does not hold for server workloads, where access patterns are highly irregular.

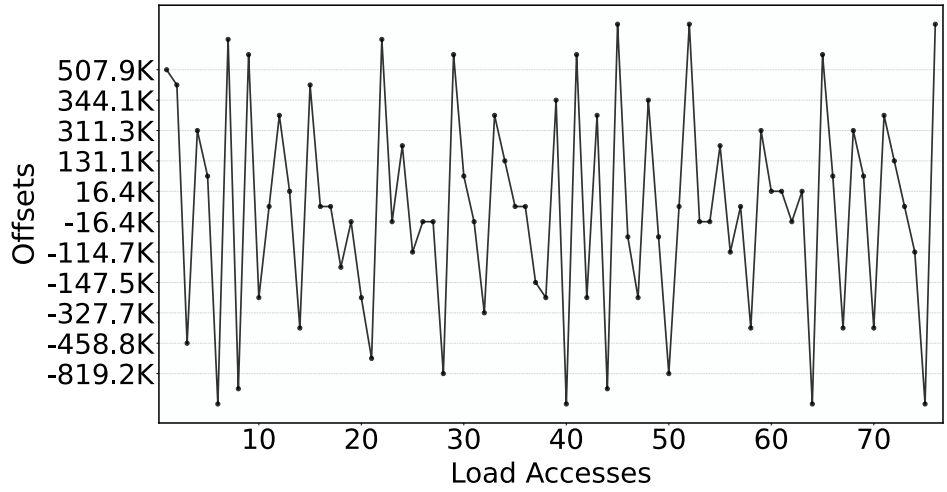


Figure 2.6: Local offsets observed for PC `0xffffb7f202f8` in the `server_011` workload. The x-axis represents 70 consecutive L1D accesses generated by the same PC.

Figure 2.6 shows that offsets for a single PC vary significantly over time and do not exhibit stable behavior. Additionally, these offsets often span distant memory regions, making pattern learning difficult and reducing prediction accuracy.

This behavior is not limited to a single instruction. Across the `server_011` workload, approximately 59.86% of PCs exhibit irregular access patterns, while only 28.64% show regular behavior. The remaining 11.50% correspond to PCs that generate only a very small number of load accesses (typically one or two), which is insufficient to infer any stable access pattern. These results indicate that irregular memory access behavior is the dominant characteristic of the workload rather than an isolated phenomenon associated with a few PCs.

Insight: PC-based prefetchers are ineffective for server workloads because most instructions exhibit irregular and non-stationary access patterns, limiting the usefulness of per-PC correlation for accurate prediction.

2.4.2 Transient Global Offsets

Global prefetchers attempt to overcome local irregularity by analyzing the overall memory access stream. However, their effectiveness is limited by *transient offsets*, which appear only

for short execution phases.

Since these prefetchers rely on epoch-based learning, offsets that seem useful in one phase often disappear in the next. At the end of each learning round, MLOP evaluates the collected offset scores and selects only those offsets that satisfy its selection thresholds. A learning round is considered *failed* when no offset is selected after this evaluation process, resulting in a prefetch degree of zero. As a result, the prefetcher may fail to identify any useful offset despite ongoing memory accesses, leading to missed prefetching opportunities. Figure 2.7 shows the percentage of such failed learning rounds across the evaluated workloads.

We further evaluate MLOP under different lookahead levels to understand the trade-off between prediction depth and hardware cost. Table 2.2 shows that increasing lookahead provides only marginal performance improvements while steadily increasing storage requirements. In fact, beyond a certain point, increasing the lookahead level leads to performance degradation. This highlights that higher lookahead levels do not necessarily translate into proportional performance gains, reinforcing the impact of transient and unstable access offsets in server workloads.

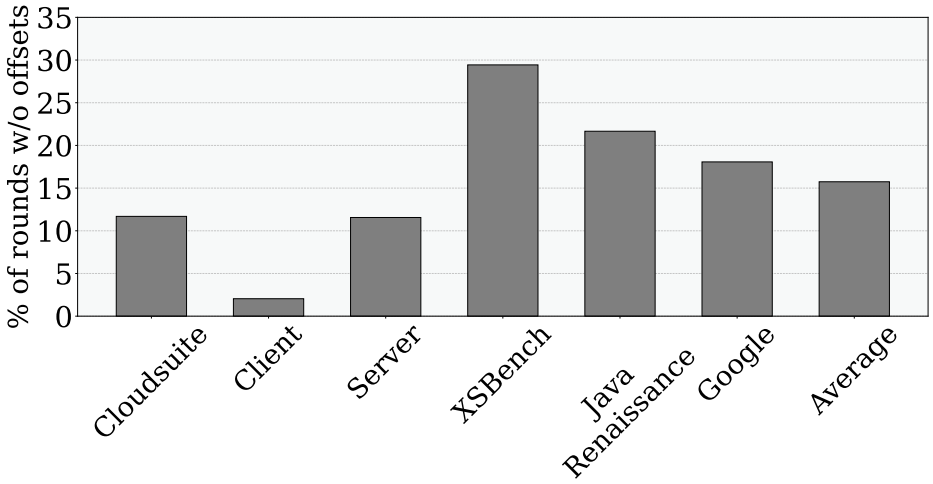


Figure 2.7: Fraction of MLOP learning rounds that fail to select any prefetch offset.

Table 2.2: MLOP performance (Speedup) and storage cost for different lookahead levels.

Lookahead Level	Speedup (in %)	Storage (KB)
4	1.34	3.4
6	1.36	5.1
8	1.30	6.8
10	1.29	8.5
12	1.22	10.2
14	1.12	11.9

Insight: Short-term, epoch-based learning fails in server workloads due to the transient nature of memory access patterns.

2.4.3 Persistent Offsets

Despite irregular and transient behavior, certain offsets persist over longer execution periods. These *persistent offsets* occur frequently and represent stable relationships in the global memory stream.

Although server workloads exhibit highly irregular memory access patterns, they often contain recurring relationships in the global memory stream. These relationships arise from the organization of data structures and their traversal patterns, such as pointer-based structures, hash tables, and graph workloads. While accesses generated by a single instruction may appear unpredictable, program execution frequently revisits similar sequences of memory locations over time. This behavior is analogous to temporal locality. For example, a sequence of accesses such as A, B, C, and D reappear multiple times during execution, even when the addresses are not contiguous and are generated by different instructions. Consequently, the corresponding address offsets also recur, causing a small set of offsets to appear consistently throughout execution and exhibit persistent behavior. Figure 2.8 shows that a small number of offsets contribute to a large fraction of memory accesses. This indicates that long-term patterns exist and can be exploited effectively.

Insight: A small set of persistent global offsets accounts for a significant portion of memory accesses.

2.4.4 Design Challenges

Designing an effective prefetcher for server workloads requires balancing multiple competing factors. The prefetcher must handle irregular patterns, adapt to changing behavior, and

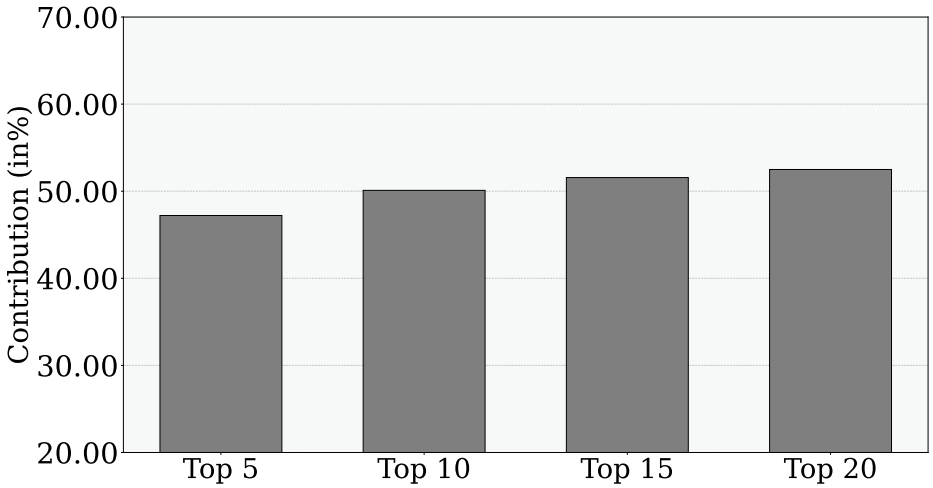


Figure 2.8: LLC access coverage achieved by the top-k most frequent (persistent) global offsets. A small number of offsets account for a large fraction of accesses.

maintain high accuracy without excessive overhead. At the same time, it must avoid unnecessary memory traffic and cache pollution while remaining efficient for large workloads and multi-core systems.

Insight: An effective design must adapt dynamically while maintaining a balance between accuracy, coverage, and overhead.

In this chapter, we first discussed the organization of the memory hierarchy and the characteristics of modern server processors and server workloads. We then reviewed several state-of-the-art hardware data prefetching techniques and highlighted their underlying design principles, strengths, and limitations. Finally, through the motivation study, we analyzed the behavior of local and global offsets, identified the impact of transient and persistent access patterns, and discussed the key challenges involved in designing an accurate and bandwidth-efficient prefetcher for multicore systems. These observations motivate the design of Toppy, which is presented in the next chapter.

Chapter 3

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

This chapter presents the design and implementation of Toppy, a lightweight bandwidth-aware offset prefetcher for multicore processors. Toppy is based on the observation that many applications exhibit globally persistent memory access offsets that can be leveraged for accurate prefetch generation. The chapter first describes the overall workflow of Toppy and the use of the Misra-Gries [36] algorithm for identifying persistent offsets. We then discuss the adaptive offset-selection mechanism, offset accuracy tracking, and the DRAM bandwidth-aware control policy used to regulate prefetch aggressiveness. Finally, we describe the implementation details and storage overhead of Toppy.

We propose *Toppy*, a lightweight last-level cache (LLC) prefetcher that exploits *persistent global offsets* to improve memory access latency. A global offset is defined as the difference (at cache-line granularity) between two consecutive LLC accesses. Unlike prior approaches that pre-select a limited set of candidate offsets [35], Toppy dynamically explores all possible offsets within a bounded range and identifies the most frequently occurring ones.

Toppy continuously monitors LLC accesses and maintains an offset tracking table to identify the top-k persistent offsets. To efficiently track these offsets under strict storage constraints, Toppy employs the Misra-Gries [36] algorithm, a streaming algorithm designed to identify frequent elements using bounded memory.

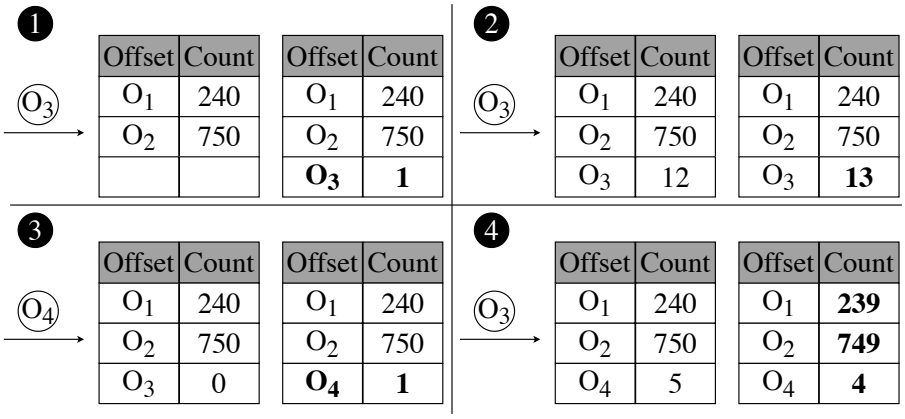


Figure 3.1: Misra-Gries algorithm in action for an offset table of size three. O_1 to O_4 are the offsets with their respective counts. Steps 1 to 4 show four possible scenarios as mentioned in Section 3.1.

3.1 Design

On every load access that reaches the LLC, Toppy computes the global offset between the current address and the previously accessed address. The computed offset is then processed by the offset table, which maintains candidate offsets along with their frequency counts. Figure 3.1 illustrates the workflow used to identify the top-k persistent offsets using the Misra-Gries algorithm. Using the top-k offsets identified by the offset table, Toppy generates up to k prefetch requests for each LLC access.

Toppy employs the Misra-Gries [36] algorithm to track the top-k most frequently occurring offsets. The algorithm operates as follows:

1. If the computed offset is not present in the offset table and the table contains a free entry, a new entry is allocated for the offset.
2. If the offset is already present in the table, its corresponding frequency counter is incremented.
3. If the table is full but contains an entry with a zero count, that entry is replaced with the new offset and its counter is initialized to one.
4. If none of the above conditions are satisfied, all counters in the table are decremented by one until at least one entry reaches zero.

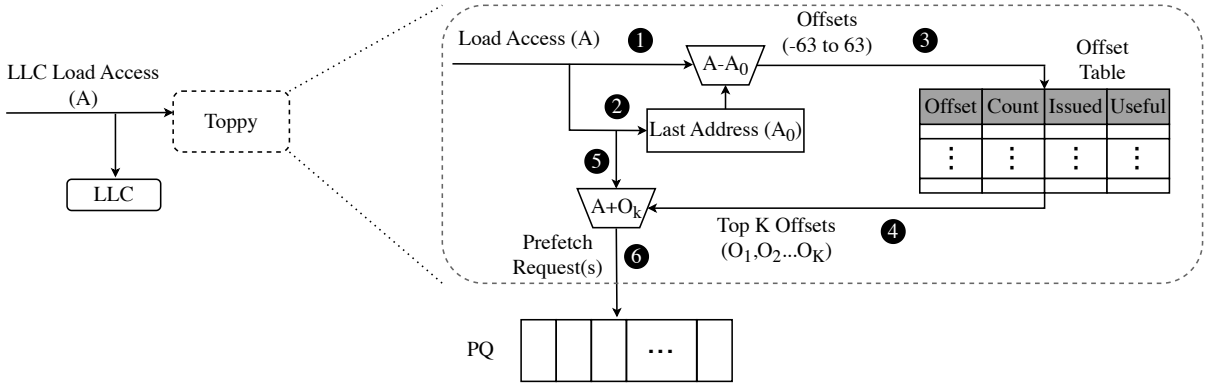


Figure 3.2: Toppy prefetcher in action at LLC. For each offset, `Count` counts the number of times the offset is seen. `Issued` counts the number of addresses generated by Misra Gries with this offset and `useful` counts the number of subsequent matches to the addresses generated by a specific offset. The ratio of issued to useful is used to quantify per offset accuracy.

Through this process, transient offsets are gradually filtered out, while frequently occurring persistent offsets retain non-zero counts and continue to remain in the table.

3.2 Toppy at the LLC in Action

Figure 3.2 illustrates the operation of Toppy at the LLC. Upon a load access to address A (❶), Toppy computes the offset relative to the previously accessed address A_0. The previous-address register is then updated with the current address (❷). The computed offset $(A - A_0)$ is inserted into the offset table, which maintains offsets within a bounded range along with their corresponding frequency counts (❸). Using the Misra-Gries algorithm, the table identifies the top-k most frequently occurring offsets $(O_1, O_2, \ldots, O_k)$ (❹). For each selected offset, Toppy generates a prefetch address of the form $A + O_i$ (❺). The generated prefetch requests are then inserted into prefetch queue (PQ) (❻).

3.3 Adaptive Prefetching Based on DRAM Traffic

Toppy selects the top-k offsets identified by the Misra-Gries algorithm, where the maximum value of k is set to 12 (Figure 2.8). While increasing k improves prefetch coverage, it can also increase DRAM traffic and reduce overall prefetch accuracy, since not all selected offsets generate useful prefetches. Similar coverage–accuracy trade-offs have been observed in prior

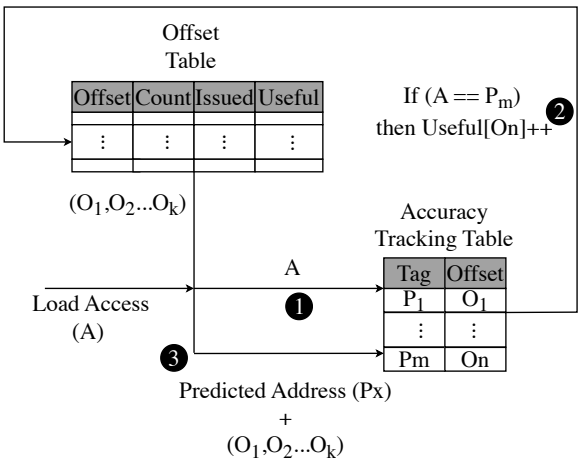


Figure 3.3: Adaptive Toppy with offset accuracy. Tag corresponds to the tag of an address.

prefetching techniques [15, 16]. To address this issue, we extend Toppy with an adaptive mechanism that dynamically selects offsets based on their runtime usefulness and current DRAM traffic conditions.

The offset table maintains the frequent offsets identified by the Misra-Gries algorithm along with their frequency counts. In addition, each offset entry maintains two counters for accuracy tracking:

- **Issued:** Number of predicted addresses generated using the offset
- **Useful:** Number of predicted addresses that later result in demand hits

To estimate the usefulness of each offset, Toppy uses an accuracy tracking table, shown in Figure 3.3. The table stores predicted addresses along with the corresponding offsets used to generate them. Figure 3.3 illustrates the workflow of offset accuracy tracking. When a future demand load access A (❶) matches an address stored in the accuracy tracking table, the corresponding *useful* counter for that offset is incremented in the offset table (❷). Simultaneously, a new set of predicted addresses generated using the current top-k offsets is inserted into the accuracy tracking table (❸), and the corresponding *issued* counters are updated.

Note that the addresses stored in the accuracy tracking table are used only for offset usefulness estimation and are not necessarily actual prefetch requests inserted into the prefetch queue (PQ). Actual prefetch requests are generated only using offsets whose accuracy exceeds the threshold.

Toppy evaluates offset accuracy over fixed-size epochs defined using LLC misses. During each epoch, the accuracy of an offset is computed as:

$$Accuracy = \frac{Useful}{Issued}$$

At the end of every epoch, offsets whose accuracy exceeds a predefined threshold are selected for prefetch generation in the subsequent epoch, while low-accuracy offsets are filtered out. The *issued* and *useful* counters are then reset to allow Toppy to adapt to changing memory access patterns, similar to prior adaptive prefetching techniques [16]. However, the offsets and their corresponding frequency counts learned by the Misra-Gries algorithm are retained across epochs, enabling Toppy to continuously track globally persistent offsets while monitoring their usefulness locally within each epoch.

At the beginning of a new epoch, none of the offsets initially satisfy the accuracy threshold because all accuracy counters are reset. To avoid losing prefetch coverage during this learning phase, Toppy temporarily continues using the top-k offsets selected in the previous epoch until sufficient runtime information is collected in the current epoch.

To further control memory bandwidth consumption, Toppy dynamically adjusts the offset-accuracy threshold based on observed DRAM traffic levels. Under high DRAM bandwidth utilization, Toppy uses a higher accuracy threshold, allowing only highly accurate offsets to generate prefetch requests and thereby reducing unnecessary DRAM traffic. Conversely, when DRAM bandwidth utilization is low, Toppy lowers the threshold to enable more offsets to participate in prefetch generation, thereby improving prefetch coverage and memory-level parallelism.

3.4 Implementation Details

We set the maximum value of k to 12 based on the ability of the selected offsets to cover LLC misses. The offset table size is set to 16 entries. The table size must be sufficiently large to avoid frequent conflicts and unnecessary replacements when the table becomes full. In particular, the design must ensure that transient offsets do not evict offsets that may eventually emerge as persistent offsets. Figure 3.4 shows the impact of offset table size on the number of conflicts per million instructions caused by a full table. We observe that an offset table with 16 entries significantly reduces conflicts while maintaining low hardware overhead.

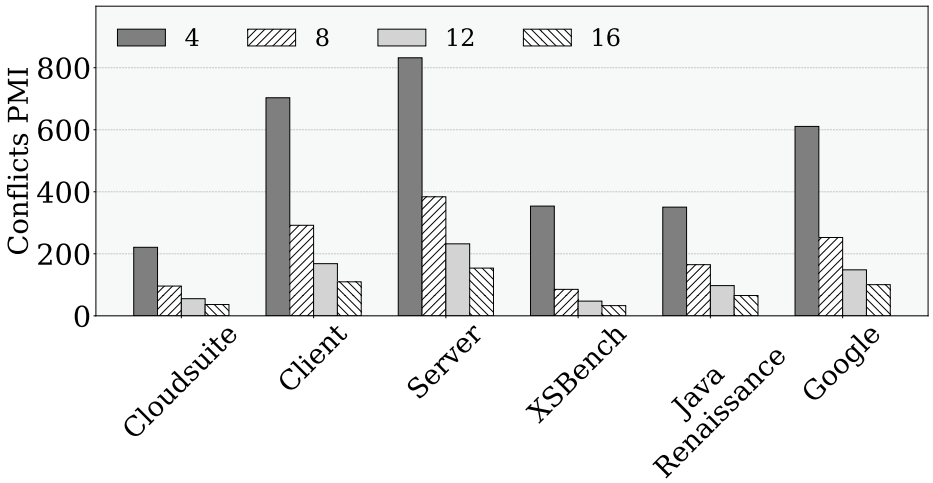


Figure 3.4: Offset table conflicts per million instructions for different table sizes at the LLC.

To evaluate the approximation introduced by the Misra-Gries algorithm, we compared the offsets selected by Toppy with those selected by an oracle implementation that maintains exact frequencies for all observed offsets. Across execution, the Misra-Gries structure achieved an average Top-k Recall of 97%, indicating that 97% of the oracle-selected top-k offsets were successfully retained by the bounded-memory tracker. This result demonstrates that the limited-size offset table preserves nearly all important persistent offsets despite its approximate nature. Overall, a 16-entry offset table is sufficient to prevent transient offsets from interfering with persistent offset tracking.

To track the usefulness of offsets and monitor available DRAM bandwidth, Toppy operates using epoch-based adaptation. We use an epoch size of 1024 LLC misses. At the beginning of each epoch, Toppy continues using the top-k offsets selected in the previous epoch for the initial 128 LLC misses to avoid loss of prefetch coverage during the learning phase. After every epoch of 1024 LLC misses, the *issued* and *useful* counters associated with each offset are reset to zero.

To monitor DRAM bandwidth utilization, we adopt the mechanism proposed in DSPatch [16]. CAS commands provide a lightweight approximation of DRAM bandwidth utilization because each CAS operation corresponds to a column read or write that transfers data on the DRAM bus. Unlike row activation or precharge commands, which only manage DRAM row state, CAS commands directly represent useful data movement and therefore closely correlate with actual memory traffic. As a result, counting CAS commands over a fixed time window provides an efficient and low-overhead mechanism for estimating DRAM bandwidth

pressure. At the DRAM controller, the number of DRAM column access (CAS) commands is counted over a window of $4 \times t_{RC}$ cycles. The measured bandwidth utilization is then quantized into utilization levels corresponding to 25%, 50%, and 75% of peak DRAM bandwidth (MTPS).Based on the observed DRAM bandwidth utilization, Toppy dynamically adjusts the offset-accuracy threshold used for top-k offset selection. When DRAM bandwidth utilization is below 25%, Toppy uses an offset-accuracy threshold of 0.1 to enable aggressive prefetching and improve coverage. As DRAM bandwidth pressure increases, the threshold is gradually increased to reduce unnecessary prefetch requests. Specifically, Toppy uses thresholds of 0.3, 0.4, and 0.5 for DRAM bandwidth utilizations of 25%-50%, 50%-75%, and greater than 75%, respectively. At the end of every epoch of 1024 LLC misses, Toppy evaluates the current DRAM bandwidth utilization and selects the offset-accuracy threshold for the subsequent epoch accordingly.

The selected threshold values are based on the trade-off between prefetch coverage, accuracy, and DRAM traffic. Lower thresholds allow a larger number of offsets to participate in prefetch generation, increasing coverage but also generating additional memory traffic. In contrast, higher thresholds restrict prefetch generation to only the most accurate offsets, improving prefetch accuracy while reducing coverage. Since the optimal operating point depends on the available DRAM bandwidth, Toppy dynamically adjusts the threshold based on observed memory traffic. The chosen threshold values were determined empirically to provide a good balance between performance improvement and memory bandwidth consumption across the evaluated workloads. A detailed sensitivity analysis of these threshold values is presented in Chapter 4.

Note: In the current implementation, recording predicted addresses generated using all top-k offsets into the accuracy tracking table may require a highly multi-ported design, which is impractical. One possible implementation approach is to extend the conventional prefetch queue (PQ) into an extended prefetch queue (X-PQ), where each entry additionally stores the offset used to generate the corresponding predicted address along with a control bit indicating whether the entry should be issued as an actual prefetch request. Entries marked for prefetching are sent to the lower-level cache or memory hierarchy, while all entries can still be inserted into the accuracy tracking table for usefulness estimation. This design enables the predicted address and its corresponding offset to be recorded in the accuracy tracking table within the same cycle in which the entry is dequeued from the X-PQ, thereby avoiding the need for a highly multi-ported accuracy tracking table while preserving offset accuracy tracking functionality.

Table 3.1: Per-core storage overhead of Toppy at LLC

Evaluation window	10-bit epoch counter (1024 LLC misses)	10 bits
Offset Table (16 entries)	7-bit offset, 23-bit frequency counter, 10-bit issued counter, 10-bit useful counter	800 bits
Accuracy Tracking Table (128 entries)	12-bit tag, 7-bit offset, 7-bit FIFO pointer	2439 bits
Selected Top-k Offsets	Up to 12 offsets (7 bits each)	84 bits
Last accessed address	46-bit address register	46 bits
Control logic	DRAM bandwidth monitoring and adaptive threshold computation	20 bits
Total		**$\approx$ 425 B**

3.4.1 Storage Overhead

Table 3.1 summarizes the per-core storage overhead of Toppy at the LLC. Overall, Toppy requires approximately 425 B of storage per core, making it significantly more lightweight than prior correlation-based prefetchers such as Bingo [13]. The storage overhead primarily consists of the offset table, the accuracy tracking table, and additional metadata required for adaptive prefetch control.

The offset table contains 16 entries, where each entry stores a 7-bit offset, a 23-bit frequency counter used by the Misra-Gries algorithm, and two 10-bit counters corresponding to the *issued* and *useful* fields used for offset accuracy tracking. We use a 10-bit epoch counter since Toppy evaluates offset usefulness over an epoch of 1024 LLC misses.

The accuracy tracking table is implemented using 128 entries. Based on our experimental evaluation, we found that 128 entries provide a good balance between storage overhead and prefetch history coverage. In the best case, the table can maintain history for up to 128 outstanding prefetched addresses, while in the worst case, when all 16 offset table entries are active, it can still retain at least eight history entries per offset. Each entry stores a 12-bit tag corresponding to the prefetched address along with the 7-bit offset used to generate that prefetch request. Instead of storing the full 52-bit physical address, Toppy employs a 12-bit tag for address comparison, significantly reducing storage overhead. Despite the potential for aliasing, the average collision and false-positive rates remain low (less than 5%). Consequently, a 12-bit tag provides a favorable storage-performance trade-off while maintaining accurate usefulness tracking. The table uses a FIFO replacement policy managed using a single 7-bit pointer to track the oldest entry in the 128-entry table.

In addition to these structures, Toppy maintains storage for the currently selected top-

k offsets, the previously accessed address used for offset computation, and small control registers required for DRAM bandwidth monitoring and adaptive threshold computation.

3.5 Why Toppy Works

Server workloads frequently execute diverse control flows, including repeated invocations of libraries and kernel routines. While individual access patterns may appear irregular, prior studies have shown that such workloads still exhibit underlying statistical regularities [14].

Toppy captures these recurring patterns by identifying the most frequent global offsets, independent of program structure. Although accesses generated by a particular instruction may not follow stable patterns, server applications repeatedly traverse similar data structures and execution paths, creating recurring relationships between memory locations. These repeated relationships manifest as persistent offsets in the global memory stream. By focusing on long-term offset frequencies rather than instruction-specific behavior, Toppy is able to learn stable prefetching opportunities that remain effective even when execution paths change.

Unlike prior approaches such as BOP and MLOP [35], which rely on short-term evaluation windows and transient offsets, Toppy leverages long-term frequency information to identify stable prefetching opportunities. This allows Toppy to achieve higher coverage despite the inherent irregularity of server workloads. Although Toppy, like most prefetchers, may exhibit moderate accuracy, its ability to capture a large fraction of memory accesses through persistent offsets enables significant overall performance improvement.

In this chapter, we presented Toppy, a lightweight offset-based hardware prefetcher that combines persistent offset learning with adaptive bandwidth-aware prefetch control. Toppy uses the Misra-Gries algorithm to identify globally persistent offsets, while dynamically filtering offsets based on runtime usefulness and DRAM bandwidth availability. The proposed design achieves adaptive prefetch generation with modest hardware overhead, making it suitable for multicore systems. The next chapter evaluates the effectiveness of Toppy across a wide range of single-core and multicore workloads.

Chapter 4

Evaluation

This chapter evaluates the performance and effectiveness of Toppy across a diverse set of workloads. We first analyze the single-core performance benefits of Toppy and compare it against several state-of-the-art hardware prefetchers. We then present sensitivity studies to understand the impact of different design parameters on performance. Finally, we evaluate the behavior of Toppy in multicore environments and analyze the interaction between adaptive prefetching, memory bandwidth utilization, and overall system performance.

We perform our evaluation using the latest version of ChampSim [6], a widely used trace-driven simulator designed for microarchitectural research. ChampSim has been extensively adopted in data prefetching competitions such as the Data Prefetching Championship (DPC) [4, 8], making it a reliable platform for evaluating prefetching techniques. Additionally, several recent proposals in prefetching and cache management have been implemented and analyzed using ChampSim [17, 13, 40, 45, 38].

We extend ChampSim with multiple enhancements to better reflect modern processor behavior. These include detailed modeling of address translation in the memory hierarchy and an accurate DRAM timing model. The baseline system configuration used in our experiments is summarized in Table 4.1, and closely resembles the architectural characteristics of Intel Granite Rapids processors [7].

For workload diversity, we utilize publicly available traces from several benchmark suites, including Cloudsuite [2], CVP-1 [3] traces provided by Qualcomm's datacenter team, Java Renaissance traces from a recent MICRO 2024 publication [44], and Google workloads released as part of DPC-4 [8]. These workloads collectively represent a wide range of modern applications spanning client, server, and cloud environments.

The configurations of the state-of-the-art hardware prefetchers used for comparison are

Table 4.1: Simulation parameters of the baseline system.

Core	Out-of-order, hashed perceptron branch predictor [29], 4 GHz with 6-issue width, 4-retire width, 352-entry ROB, FTQ: 24 entries (192 instructions)
TLBs	L1 iTLB/dTLB: 64 entries, 4-way, 1 cycle; STLB: 2048 entries, 16-way, 8 cycles
MMU Caches	2-entry PSCL5, 4-entry PSCL4, 8-entry PSCL3, 32-entry PSCL2, searched in parallel, one cycle
L1I	32 KB, 8-way, 4 cycles, LRU, FDIP [43] prefetcher
L1D	48 KB, 12-way, 5 cycles, LRU, IP-Stride prefetcher
L2	2 MB, 16-way, 15 cycles, SRRIP [28], non-inclusive
LLC	3 MB/core, 12-way, 74 cycles, DRRIP [28], non-inclusive
MSHRs	16/16/48 at L1I/L1D/L2, 64/core at the LLC
DRAM controller	One channel/4-cores, DDR5-6400 MTPS, FR-FCFS, 64-entry RQ and WQ, reads prioritized over writes, write watermark: 7/8th
DRAM chip	4 KB row-buffer per bank, open page, burst length 16, t_{RP}: 12.5 ns, t_{RCD}: 12.5 ns, t_{CAS}: 12.5 ns

listed in Table 4.2. Each simulation executes 200 million instructions after an initial warmup phase of 50 million instructions to ensure steady-state behavior. In multi-core experiments, simulations continue until the slowest core completes execution.

Performance evaluation differs for single-core and multi-core settings. For single-core systems, we measure improvements in Instructions Per Cycle (IPC). For multi-core systems, we use weighted speedup (WS) [22] and harmonic mean of speedups (HS) [21, 41]. While WS captures overall throughput, HS emphasizes fairness across applications. These metrics are defined as:

$$WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$$

$$HS = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}$$

$$Unfairness = \frac{max(IS_0, IS_1, ...IS_{N-1})}{min(IS_0, IS_1, ...IS_{N-1})}$$

$$IS(i) = \frac{CPI_i^{together}}{CPI_i^{alone}}$$

Table 4.2: Configurations of evaluated prefetchers

SPP-PFF [17]	256-entry ST, 512-entry 4-way PT, 8-entry GHR, Perceptron weights with the following entries: 4096×4, 2048×2, 1024×2, and 128×1 entries, 1024-entry prefetch table, 1024-entry reject table
BOP [35]	256-entry RR Table, 100-update, 54 offsets
Gaze [19]	64-entry 4-way FT, 64-entry 4-way AT, 256-entry 8-way PHT, 32-entry 8-way PB, 8-entry DPCT
Berti [38]	8-entry 16-way HT, 16-entry Delta Table
Bingo [13]	2 KB region, 64/128/4K-entry FT/AT/PHT
MLOP [45]	128-entry AMT, 500-update, 16-degree
IPCP [40]	128-entry IP table, 8-entry RST table, and 128-entry CSPT table
Pythia [15]	2-vault QVStore with 3 planes per vault, each plane having 128×16 entries, and a 256-entries EQ
KPC-P [34]	256-entry Signature Table, 512-entry Pattern Table, 64-entry LLC Sampler

Here, IPC_i^{together} represents the IPC of application i when co-executed with other applications on an N-core system, while IPC_i^{alone} denotes its IPC when running in isolation. The unfairness metric quantifies performance imbalance by comparing the maximum and minimum slowdowns experienced across applications.

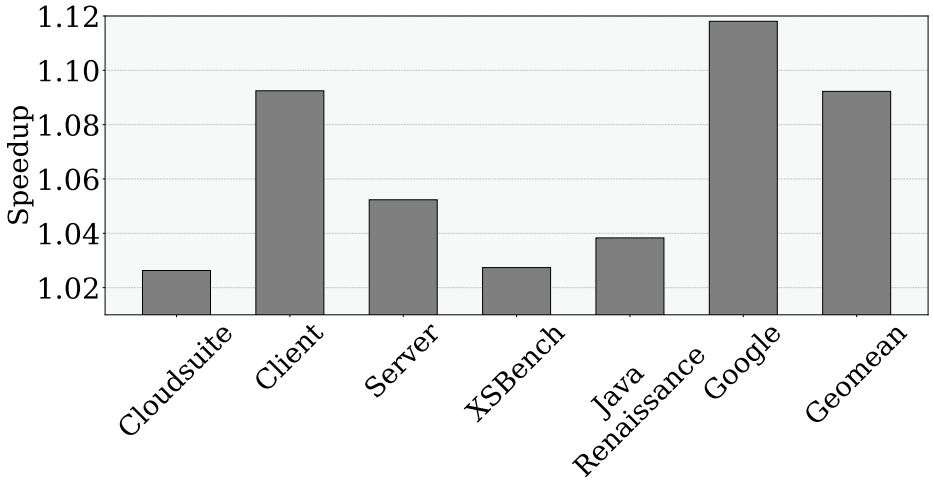


Figure 4.1: Performance of Toppy prefetcher on server workloads normalized to L1 IP-stride

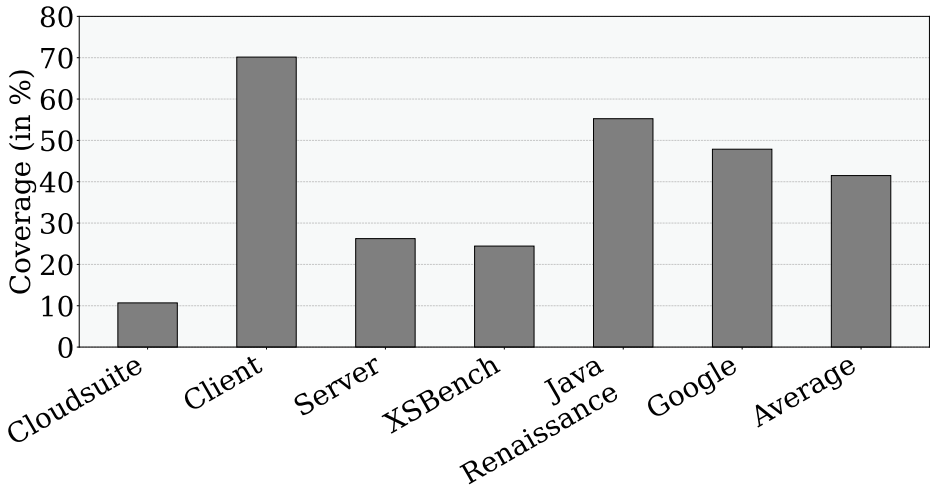


Figure 4.2: Prefetch coverage at the LLC with the Toppy prefetcher.

4.1 Single-core performance

Figure 4.1 presents the performance gains achieved by deploying Toppy at the LLC. On average, Toppy improves performance by 9.22% over the baseline configuration that uses an IP-stride prefetcher at the L1D. This improvement highlights Toppy's ability to capture memory access patterns beyond simple stride-based behavior. Among all evaluated workloads, Google traces benefit the most, achieving an average speedup of approximately 11.8%. Across the entire workload suite, Toppy consistently delivers at least a 2% improvement over the baseline. However, XSBench shows relatively smaller gains due to the absence of sufficient offsets that satisfy the accuracy threshold of 0.1, limiting Toppy's effectiveness.

Prefetch coverage and prefetch degree Figure 4.2 shows the proportion of LLC misses that are successfully covered by Toppy. On average, Toppy covers 41.48% of LLC misses, demonstrating its effectiveness in predicting future memory accesses. Client workloads from `CVP1` achieve the highest coverage, reaching approximately 70%. Figure 4.3 illustrates the average number of offsets (K) selected by Toppy. Across workloads, Toppy typically operates with a prefetch degree of five, selecting the top five most persistent offsets. In contrast, for XSBench, the lack of high-confidence offsets results in a reduced prefetch degree of two.

DRAM traffic and prefetch accuracy Table 4.3 summarizes the impact of different offset accuracy thresholds on prefetch accuracy, performance, and DRAM traffic. A lower threshold favors aggressive prefetching, increasing coverage and improving performance, but at the cost of additional memory traffic. A threshold of 0 effectively disables accuracy filter-

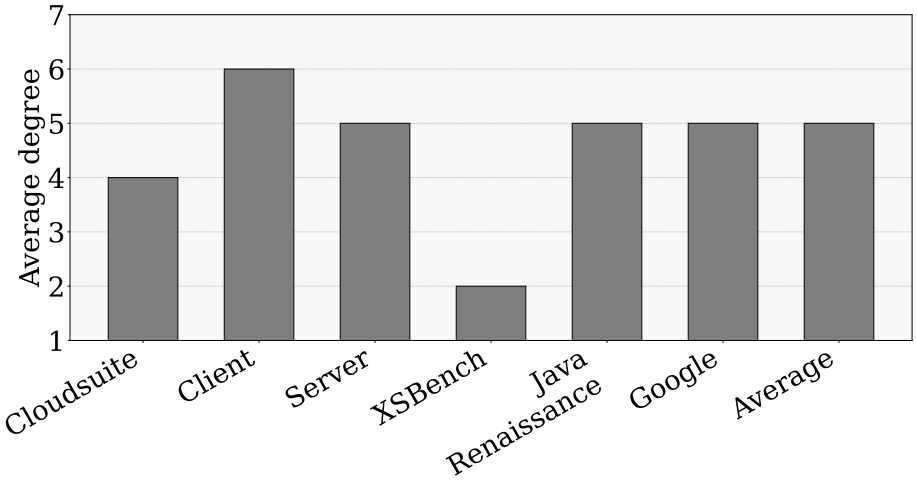


Figure 4.3: Average value of K with Toppy at LLC

Table 4.3: Toppy's prefetch accuracy, speedup, and DRAM traffic based on offset accuracy threshold. DRAM TPKI of the baseline IP-stride 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

ing and corresponds to the baseline Misra-Gries design, achieving the highest performance improvement of 10.68%. However, this gain comes at the expense of substantial DRAM traffic, reaching 30.5 TPKI (Traffic per Kilo Instructions).

As the threshold increases from 0.1 to 0.5, prefetch accuracy improves from 34.84% to 49.57%, indicating that the prefetcher becomes more selective. However, this increased selectivity reduces performance gains from 9.22% to 4.33%, as fewer useful prefetch opportunities are exploited. At the same time, DRAM traffic decreases significantly from 19.54 to 13.63 TPKI. These results highlight the fundamental trade-off between performance and memory bandwidth utilization: lower thresholds maximize performance through aggressive prefetching, whereas higher thresholds improve accuracy and reduce memory traffic at the expense of performance.

Note: The bandwidth-aware adaptation mechanism has no impact on the single-core results because memory bandwidth is not a performance bottleneck in this configuration.

Although lower accuracy thresholds improve performance by increasing prefetch coverage, Toppy's overall prefetch accuracy remains relatively modest. Deploying Toppy at the L1D or L2 cache levels therefore introduces cache pollution, which increases L1D DMPKI as well as data and instruction MPKI in the L2 cache. The resulting increase in cache contention can exacerbate front-end bottlenecks and ultimately degrade performance. In contrast, the larger capacity of the LLC provides greater tolerance to inaccurate prefetches, while most instruction working sets remain resident in the upper cache levels. Consequently, Toppy is deployed at the LLC, where it can exploit memory-level parallelism while minimizing adverse effects on the front-end. We acknowledge that a more accurate version of Toppy could potentially be deployed at the L1D or L2 levels, which represents an interesting direction for future work.

Table 4.4: Top-k offsets chosen by Toppy.

Benchmark	**Offsets**	**Fraction of offsets in %**
Cloudsuite	-8, -1, 1, 2, 4, 5, 9, 13, 42, 43, 60, 62	75.24
Client	-5, -3, -2, -1, 1, 2, 3, 4, 5, 7	76.10
Server	-9, -6, -5, -3, -2, -1, 1, 2, 3, 4, 5, 6, 11	75.15
XSBench	-15, -2, -1, 1, 2, 3, 4, 5, 6, 7	76.10
Java Renaissance	-9, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6, 7, 10	75.3
Google	-12, -7, -6, -5, -4, -2, -1, 1, 2, 3, 4, 5, 7, 12	75.10

There is a clear trade-off between accuracy and DRAM traffic. Lower thresholds result in more aggressive prefetching, increasing DRAM traffic, while higher thresholds reduce unnecessary memory accesses. Despite this increase, overall bandwidth utilization remains low (less than 0.5%) due to the high bandwidth of DDR5-6400 memory. Without prefetching, bandwidth utilization is approximately 0.12%. Given these observations, Toppy selects a threshold of 0.1 to maximize performance, although a threshold of 0.4 provides a balanced trade-off.

Top-k selected offsets. Table 4.4 lists the offsets selected by Toppy that collectively account for more than 75% of all selected offsets. Most of these offsets fall within the range of -16 to 16, indicating strong spatial locality in memory accesses across workloads.

4.2 Comparison with state-of-the-art prefetchers

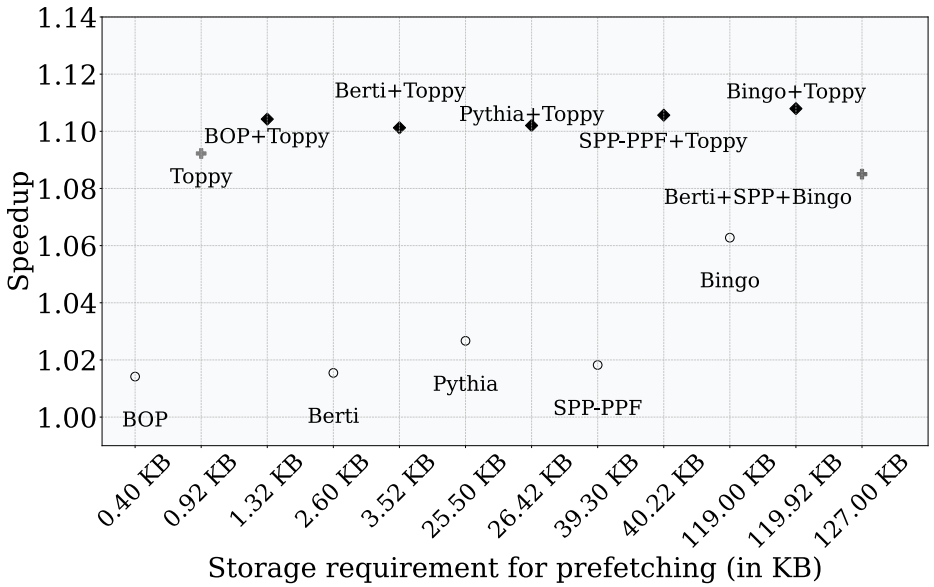


Figure 4.4: Speedup with Toppy and other state-of-the-art prefetchers, including the best combination of prefetchers (Berti+SPP+Bingo) normalized to an L1D IP-stride prefetcher.

Performance vs storage. Figure 4.4 compares Toppy with existing prefetchers in terms of performance and storage overhead. Toppy achieves the highest performance improvement of 9.22% while requiring less than 1KB of storage. In comparison, Bingo achieves 6.28% improvement but requires over 100KB of storage. Interestingly, Toppy also outperforms a combined approach using Berti, SPP, and Bingo, which achieves slightly above 8% improvement. When used alongside other prefetchers such as BOP, Bingo, and IPCP, Toppy further increases performance to approximately 11%.

Performance vs DRAM traffic. Figure 4.5 compares performance improvements against DRAM traffic for different prefetchers. With an accuracy threshold of 0.5, Toppy achieves competitive performance with minimal increase in DRAM traffic, comparable to Berti.

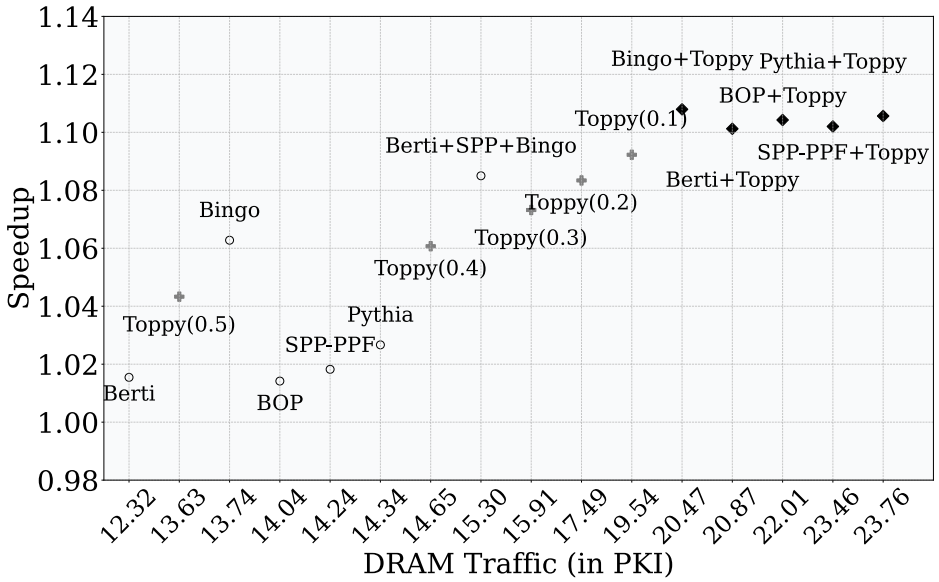


Figure 4.5: Speedup of Toppy and other state-of-the-art prefetchers, including the best combination (Berti + SPP + Bingo), normalized to an L1D IP-stride prefetcher, with their respective DRAM traffic

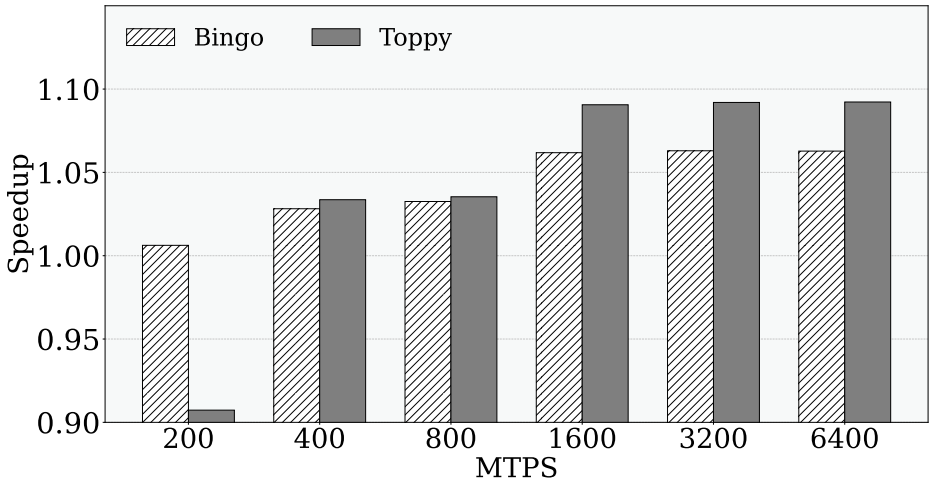


Figure 4.6: Speedup of Toppy prefetcher across different DRAM bandwidths (in MTPS).

At a threshold of 0.4, Toppy achieves performance similar to Bingo with only slightly higher DRAM traffic (14.65 TPKI compared to 13.75 for Bingo). We further evaluate performance under varying DRAM bandwidths ranging from 200 MTPS to 6400 MTPS. As shown in Figure 4.6, Toppy outperforms Bingo across most bandwidth configurations, except at 200 MTPS. Under extremely constrained bandwidth, Toppy with a threshold of 0.5 still achieves

a performance gain of 1.2%, outperforming other prefetchers such as Bingo.

4.3 Sensitivity Studies

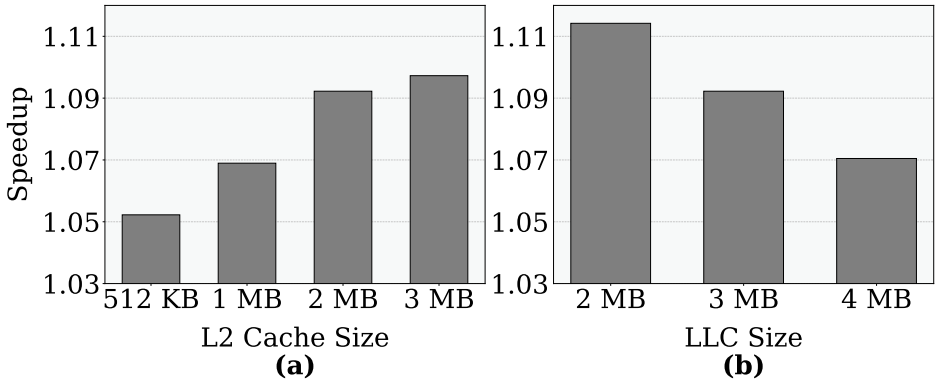


Figure 4.7: Performance sensitivity of Toppy to different L2 and LLC cache sizes, normalized to the corresponding baseline configurations.

Figure 4.7 evaluates the impact of cache hierarchy configurations on Toppy. Increasing the LLC size reduces the number of LLC misses and consequently decreases the opportunity for prefetching, limiting the achievable performance gains. Conversely, reducing the L2 cache size increases instruction cache misses (IMPKI), shifting the performance bottleneck toward the front-end and reducing the effectiveness of data prefetching. Despite these changes, Toppy consistently improves performance across a wide range of cache configurations, demonstrating robustness to variations in cache capacity.

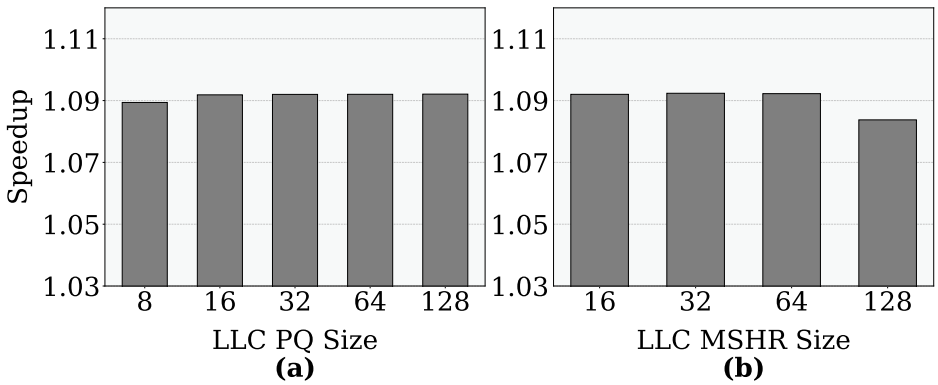


Figure 4.8: Performance sensitivity of Toppy to LLC prefetch queue (PQ) and MSHR sizes, normalized to the corresponding baseline configurations.

Figure 4.8 examines the impact of LLC Miss Status Holding Registers (MSHRs) and prefetch queue (PQ) capacity. Toppy maintains its effectiveness even when these resources are reduced because its average prefetch degree remains relatively low, averaging approximately five prefetches. Increasing the number of MSHRs or PQ entries beyond the baseline configuration provides little additional benefit, indicating that Toppy does not place excessive pressure on memory-system resources.

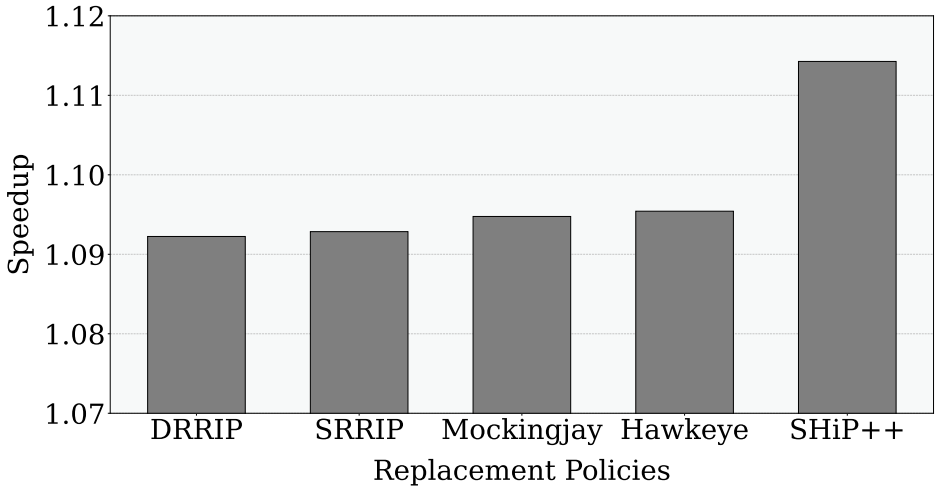
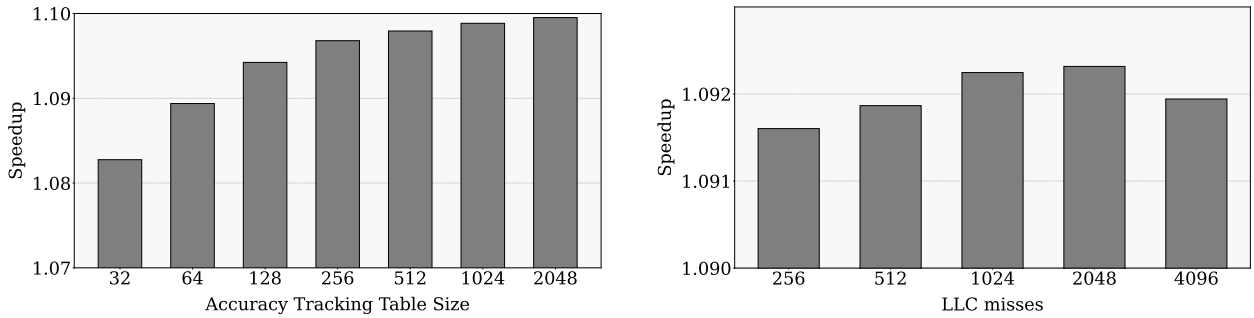


Figure 4.9: Performance sensitivity of Toppy to different LLC replacement policies, normalized to an L1D IP-stride prefetcher using DRRIP at the LLC.

Figure 4.9 evaluates Toppy under different LLC replacement policies. Across all evaluated policies, Toppy consistently achieves more than 9% performance improvement. The highest performance is observed with SHiP++ [48], which more effectively mitigates cache pollution caused by inaccurate prefetches. These results indicate that Toppy's benefits are largely independent of the underlying cache replacement strategy.



(a) Accuracy tracking table size.

(b) Epoch interval (LLC misses).

Figure 4.10: Sensitivity of Toppy to internal design parameters.

Figure 4.10 evaluates the sensitivity of Toppy's internal design parameters. Increasing the size of the accuracy tracking table improves performance by enabling more accurate estimation of prefetch usefulness; however, the benefit saturates beyond 128 entries. Similarly, increasing the epoch interval initially improves performance by providing more stable accuracy measurements, but performance begins to decline when the interval exceeds 2048 LLC misses. Based on these observations, a 128-entry tracking table and an epoch interval of 1024 LLC misses provide a favorable balance between performance and hardware overhead.

4.4 Multi-core performance

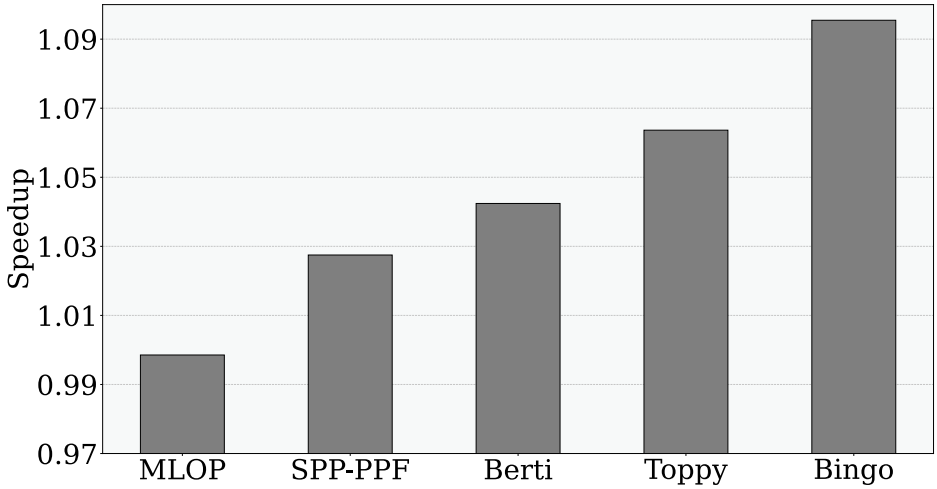


Figure 4.11: Performance improvement (in terms of weighted speedup) across 70 mixes. Bingo is the best prefetcher for multi-core, and Toppy is the second best.

Figure 4.11 summarizes the multi-core (4-core) performance results across 70 multi-programmed workload mixes. In terms of weighted speedup, Toppy achieves an average improvement of 6.4%, making it the second-best prefetcher after Bingo, which achieves 9.5%. Similar trends are observed for harmonic speedup (Figure 4.12), where Toppy improves performance by 6.6% compared to Bingo's 10.4%. Despite its lower average performance, Toppy outperforms Bingo by more than 10% in several workload mixes, demonstrating its effectiveness for certain memory access characteristics.

We also evaluate fairness using the unfairness metric. Bingo achieves the lowest unfairness value of 1.25, while Toppy achieves a comparable value of 1.30, indicating only a modest impact on fairness. In contrast, the baseline IP-stride prefetcher exhibits the highest

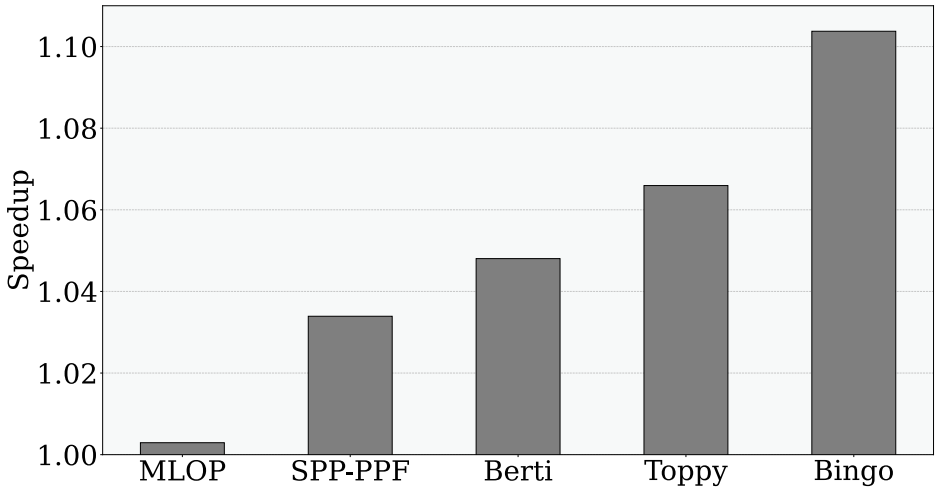


Figure 4.12: Performance improvement (in terms of Harmonic speedup) across 70 mixes. Bingo is the best prefetcher for multi-core, and Toppy is the second best.

unfairness (1.40). Overall, these results show that Toppy provides competitive multi-core performance while maintaining fairness close to the best-performing prefetcher. Overall, Bingo achieves the highest performance but requires significantly higher storage (952 KiB for an 8-core system). In contrast, Toppy achieves competitive performance with only 2.15 KiB of storage, making it a highly storage-efficient design.

Bandwidth-aware adaptation is expected to become more important in multicore environments due to increased contention for shared memory resources. In our evaluation, a static accuracy threshold of 0.1 achieves approximately ∼5% average performance improvement in the multicore setting, and performance consistently decreases as the threshold is increased further. This suggests that the workloads benefit more from maintaining prefetch coverage than from aggressively filtering prefetch requests.

Nevertheless, bandwidth-aware adaptation provides additional benefit by dynamically adjusting prefetch aggressiveness based on runtime memory pressure, increasing the average performance improvement from approximately 5% to 6.4%. While the gains are modest for the evaluated workload mixes, the mechanism helps balance coverage and bandwidth consumption without requiring a fixed threshold. We expect its benefits to become more pronounced in systems with higher memory contention, larger core counts, or more bandwidth-intensive workloads.

In this chapter, we evaluated Toppy across both single-core and multicore workloads and compared its performance with existing state-of-the-art prefetchers. The results show that

Toppy effectively improves prefetch coverage while maintaining accuracy and controlling unnecessary DRAM traffic through its adaptive bandwidth aware design. We also presented sensitivity studies to justify the design choices used in Toppy and analyzed its behavior under different workload characteristics. Overall, the evaluation demonstrates that Toppy provides an efficient and lightweight prefetching solution for modern multicore systems.

Chapter 5

Conclusion

In this work, we presented *Toppy*, an LLC-based offset prefetcher specifically designed for modern server-class workloads that are characterized by large instruction footprints and complex memory access behavior. A key observation driving the design of Toppy is that such workloads exhibit *persistent memory access offsets* that remain stable over long execution periods. Toppy effectively leverages this property to improve data prefetching decisions at the last-level cache (LLC).

To identify these persistent offsets efficiently and with minimal hardware overhead, Toppy employs the Misra-Gries heavy-hitter algorithm, enabling it to dynamically track the most frequently occurring offsets in the memory access stream. By focusing only on the most significant offsets, Toppy achieves a balance between accuracy and storage efficiency. Furthermore, we propose an adaptive mechanism that tunes the offset selection threshold, allowing Toppy to navigate the trade-off between prefetch coverage, accuracy, and DRAM traffic based on workload characteristics and system constraints.

Our evaluation demonstrates that Toppy incurs a very small per-core storage overhead of only 425 bytes, making it highly practical for real-world deployment. Despite this lightweight design, Toppy delivers significant performance improvements. In single-core configurations, it consistently outperforms all evaluated state-of-the-art prefetchers for server workloads with large instruction footprints, achieving substantial speedups while maintaining manageable DRAM bandwidth usage. On average, Toppy is able to cover approximately 40% of LLC misses, highlighting its effectiveness in capturing recurring memory access patterns.

In multi-core environments, where contention for shared resources such as LLC capacity and DRAM bandwidth becomes critical, Toppy continues to provide competitive performance. While it outperforms most existing prefetching techniques, it is slightly outper-

formed by Bingo, which utilizes significantly higher storage resources. Nevertheless, Toppy achieves a favorable balance between performance and hardware cost, making it an attractive alternative in resource-constrained systems.

Chapter 6

Acknowledgements

I would like to sincerely thank my advisor, **Prof. Biswabandan Panda**, for his constant guidance and support throughout my time at this institution. His deep knowledge, constructive feedback have been invaluable in shaping this research work.

I am also grateful to my colleagues and friends in the **CASPER** group, particularly **Shubham Roy** and **Nishkarsh Gautam**. The many discussions, idea exchanges, and collaborative learning experiences we shared have significantly contributed to refining this work and broadening my understanding.

Naman Sharma
IIT Bombay

Bibliography

[1] "The 2nd data prefetching championship (dpc-2)," Jun. 2015. [Online]. Available: https://comparch-conf.gatech.edu/dpc2/

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

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

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

[5] "Spec cpu 2017 traces for champsim," https://hpca23.cse.tamu.edu/champsim-traces/speccpu/index.html, Feb. 2019.

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

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

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

[9] Advanced Micro Devices, Inc., "Amd epyc processors," 2023, https://www.amd.com/en/processors/epyc.

[10] S. Ainsworth and L. Mukhanov, "Triangel: A high-performance, accurate, timely on-chip temporal prefetcher," in *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. IEEE, 2024, pp. 1202–1216. [Online]. Available: https://doi.org/10.1109/ISCA59077.2024.00090

[11] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 462–473. [Online]. Available: https://doi.org/10.1145/3307650.3322234

[12] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Moseley, P. Ranganathan, B. Robatmili, and A. Sabu, "AsmDB: Understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 462–473.

[13] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[14] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 52:1–52:29, 2019. [Online]. Available: https://doi.org/10.1145/3312740

[15] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *54th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 1121–1137.

[16] R. Bera, A. V. Nori, , O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *52nd Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2019, pp. 531–544.

[17] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–13.

[18] Z. Chen, C. Wu, Y. Gu, R. Jia, J. Li, and M. Guo, " Gaze into the Pattern: Characterizing Spatial Patterns with Internal Temporal Correlations for Hardware Prefetching ," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar.

2025, pp. 173–187. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA61900.2025.00024

[19] ——, "Gaze into the pattern: Characterizing spatial patterns with internal temporal correlations for hardware prefetching," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 173–187.

[20] Y. Cui, W. Chen, X. Cheng, and J. Yi, "Hyperion: A highly effective page and pc based delta prefetcher," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 4, Nov. 2024. [Online]. Available: https://doi.org/10.1145/3675398

[21] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *42nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 316–326.

[22] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, "Symbiotic job scheduling on the ibm power8," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 669–680.

[23] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 37–48.

[24] S. He, Z. Wang, X. Tang, Q. Sun, and D. Dong, "Chimera: Leveraging hybrid offsets for efficient data prefetching," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 144–155. [Online]. Available: https://doi.org/10.1145/3656019.3689613

[25] Hewlett Packard Enterprise, "What is a workload?" https://www.hpe.com/us/en/what-is/workload.html, 2026, accessed: April 2026.

[26] IBM Think, "What is a workload?" https://www.ibm.com/think/topics/workload, 2025, accessed: April 2026.

[27] Intel Corporation, "Intel xeon processor family," 2023, https://www.intel.com/content/www/us/en/products/details/processors/xeon.html.

[28] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *37th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2010, pp. 60–71.

[29] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 197–206.

[30] V. Kalbande, H. J. Deshmukh, A. Ros, and B. Panda, "Icarus: Criticality and reuse based instruction caching for datacenter applications," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 978–992. [Online]. Available: https://doi.org/10.1145/3779212.3790175

[31] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.

[32] J. Kim, P. V. Gratz, and A. L. N. Reddy, "Lookahead prefetching with signature path," in *2nd Data Prefetching Championship*, Jun. 2015.

[33] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *49th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 60:1–60:12.

[34] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, p. 737–749, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093337.3037701

[35] P. Michaud, "Best-offset hardware prefetching," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 469–480.

[36] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[37] N. P. Nagendra, B. R. Godala, I. Chaturvedi, A. Patel, S. Kanev, T. Moseley, J. Stark, G. A. Pokam, S. Campanoni, and D. I. August, "Emissary: Enhanced

miss awareness replacement policy for l2 instruction caching," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589097

[38] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *55th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2022, pp. 975–991.

[39] G. Ottoni, "HHVM JIT: A profile-guided, region-based compiler for PHP and Hack," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 151–165.

[40] 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.

[41] B. Panda, "Spac: A synergistic prefetcher aggressiveness controller for multi-core systems," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3740–3753, 2016.

[42] phoenixNAP IT Glossary, "What is server workload?" https://phoenixnap.com/glossary/server-workload, 2025, accessed: April 2026.

[43] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *32nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 16–27.

[44] D. Schall, A. Sandberg, and B. Grot, "The last-level branch predictor," in *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*. IEEE, 2024, pp. 464–479. [Online]. Available: https://doi.org/10.1109/MICRO61859.2024.00042

[45] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in *The 3rd Data Prefetching Championship*, Jun. 2019.

[46] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 996–1008.

[47] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 449–461.

[48] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*, 2017.