Zion: A Comprehensive, Adaptive, and Lightweight Hardware Prefetcher

Vadim Biryukov[1,2], Xiaoyang Lu[1], Zirui Liu[3], Kaixiong Zhou[4], Xian-He Sun[1]
[1]Illinois Institute of Technology [2]Lewis University [3]University of Minnesota [4]North Carolina State University

***Abstract*—As the gap between processor and memory performance widens, optimizing data access performance becomes increasingly critical. Hardware prefetching is a widely used technique to hide long-latency off-chip memory accesses, but state-of-the-art prefetchers struggle with diverse and dynamic access patterns. Their limited adaptability leads to excessive storage overhead and reduced effectiveness under memory-intensive workloads. We propose Zion, a comprehensive, adaptive, and lightweight hardware prefetcher for memory-intensive workloads. At its core, Zion uses Independent Temporal-Spatial Modules (ITSM) for broad pattern coverage and runtime adaptability to diverse memory access patterns. Moreover, Zion leverages runtime feedback to dynamically guide prefetching decisions and maintain efficiency under memory pressure. Extensive multi-core evaluations show that Zion consistently outperforms state-of-the-art prefetchers, achieving up to 43.2% performance improvement on SPEC and 43.0% on self-attention workloads, while maintaining low overhead and broad effectiveness.**

I. Introduction

The unbalanced technological advancements between processor speed and memory performance over the past decades have led to the well-known *Memory Wall* problem [1]. Modern memory-intensive workloads further exacerbate this problem, placing memory access latency at the forefront of performance bottlenecks [2]. Recently, the emergence of Large Language Models (LLMs) has introduced attention-based workloads, representing an important new category of memory-intensive workloads. Although traditionally executed on GPUs, there is growing interest in enabling local CPU-based LLM inference [3], [4] due to demands for privacy-preserving and low-latency execution on widely available personal devices. Additionally, advancements in compact LLM architectures, which significantly reduce model size while maintaining competitive accuracy, have made CPU-based deployments increasingly practical. Nevertheless, executing attention-based computations on CPUs poses significant challenges due to the quadratic memory complexity inherent in self-attention mechanisms [5], which greatly intensifies data movement between on-chip cache hierarchies and off-chip DRAM.

Hardware prefetching is a widely used technique to mitigate memory access latency by predicting and fetching data closer to the CPU before it is needed, thereby improving overall system performance [6]–[16]. Despite numerous proposed hardware prefetchers, we observe three critical limitations that constrain their effectiveness in memory-intensive workloads. First, memory-intensive workloads typically exhibit distinct and dynamic phases, each characterized by unique memory access behaviors. However, existing prefetchers often fail to adequately capture the diverse memory access patterns occurring across these varying phases, thereby reducing overall pattern coverage. Second, most prefetchers lack the ability to utilize runtime feedback, leading to limited adaptability to system dynamics and thus suboptimal performance. Hardware prefetches without proper runtime control can introduce undesirable side effects such as increased memory bandwidth consumption, cache pollution, and memory access interference, ultimately diminishing or even negating the expected performance benefits. Third, state-of-the-art prefetchers typically rely on a large, unified prediction table to maintain high accuracy, incurring excessive storage overhead, particularly at cache levels. Moreover, this unified prediction table inherently suffers from entry conflicts across unrelated data streams, causing premature eviction of locally frequent yet globally rare patterns, or the underestimation of infrequent but highly accurate predictions when count-based confidence mechanisms are employed. These limitations motivate the need for a comprehensive, adaptive, and lightweight hardware prefetcher for modern memory-intensive workloads.

To this end, we propose Zion, which incorporates the following core features to overcome existing limitations and effectively handle diverse and dynamic access patterns. **First**, at the core lies the ITSM (Independent Temporal-Spatial Modules) design, consisting of five prediction modules, each with a differently sized prediction table that retains entries capturing repeating access patterns to effectively exploit either temporal or spatial locality. ITSM addresses the limitations of traditional prefetchers [6]–[9], which typically rely on one global unified table prone to frequent entry conflicts. **Second**, Zion employs dynamic runtime adjustment for each ITSM module, to adjust prefetching decisions and the insertion destinations of prefetched blocks based on real-time feedback. By continuously monitoring each module's prediction accuracy and incorporating hardware feedback, Zion optimizes performance while minimizing memory pressure. **Finally**, Zion achieves a balanced trade-off between performance and overhead, reducing storage requirements while guaranteeing coverage capabilities.

The performance of Zion is extensively evaluated using memory-intensive workloads from the SPEC CPU2006 [17] and SPEC CPU2017 [18] benchmark suites, as well as diverse self-attention workloads from various LLM architectures [19] implemented in PyTorch, with memory access behavior shaped by CPU libraries and runtime backends [20]. The results demonstrate that Zion, with its comprehensive and adaptive design, consistently outperforms state-of-the-art (SOTA) hardware prefetchers, including IPCP [7], Bingo [9], SPP [6], and

Berti [8], across a range of system configurations. Notably, in a 4-core configuration, Zion achieves performance improvements of 18.2%, 10.1%, 8.0% and 5.1% over these SOTAs, respectively in SPEC CPU [17], [18] workloads. Moreover, Zion improves the performance of self-attention workloads, up to 42.2% speedup over the baseline in an 8-core configuration.

The contributions of this paper are summarized as follows:

- We comprehensively analyze the memory access patterns of memory-intensive workloads from SPEC CPU benchmarks and self-attention mechanisms, identifying their diverse and irregular behaviors.
- We propose ITSM, a novel design based on independent temporal-spatial modules, to enhance prediction accuracy and coverage of varying memory access patterns.
- We propose a runtime adaptive mechanism that dynamically adjusts prefetch and insertion decisions based on real-time system feedback.
- We extensively evaluate and demonstrate that Zion consistently outperforms SOTA prefetchers on memory-intensive workloads with lightweight overhead.

II. Background and Motivation

A. State-of-the-Art Hardware Prefetchers

Hardware prefetching is a widely adopted technique for mitigating memory access latency by proactively fetching data before it is requested, with state-of-the-art prefetchers employing diverse and distinct design strategies.

IPCP [7] classifies instruction pointers into categories: constant stride, complex stride, and global stream, enabling tailored prefetching strategies for each pattern type. IPCP employs a unified IP (instruction pointer) table shared among these categories, which can cause early evictions of infrequent but potentially valuable entries due to conflicts. **Bingo** [9] enhances spatial pattern learning by associating short-range (IP-based) and long-range (IP plus memory region) access correlations. Bingo utilizes a large (114KB) unified hardware table to accommodate and manage multiple prediction streams, improving performance at the cost of significantly increased storage overhead. **SPP** [6] predicts irregular L2 strides by encoding access patterns into signatures derived from consecutive memory addresses. Through a recursive lookahead mechanism, it issues prefetches until the confidence level drops below a threshold. However, its count-based confidence mechanism inherently underestimates entries. **Berti** [8] tracks local deltas to generate timely and precise prefetches. By associating first IP accessing a page with local deltas of that page, it maintains two compact history tables (totaling 19.85KB), resulting in moderate storage overhead. However, its count-based confidence mechanism ignores rare but accurate stream patterns in favor of frequent deltas reducing overall effectiveness.

B. Workload Memory Access Behavior

Memory-intensive SPEC Workloads. We analyze memory-intensive SPEC workloads and identify two critical challenges in SOTA prefetcher designs. First, workloads exhibit distinct and dynamic phases, each with its own unique memory access behavior. Existing prefetchers often struggle to adapt quickly to such phase variations, resulting in delayed or inaccurate predictions. Second, independent data streams frequently share the same instruction pointer (IP), causing conflicts that reduce the prediction accuracy of IP-based prefetchers.

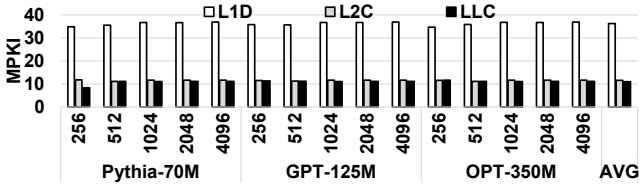

Fig. 1: MPKI across different cache levels for self-attention workloads.

For instance, in memory-intensive workloads such as *mcf*, we observe two distinct phases. The first phase involves high latencies between consecutive memory accesses, characterized by significant IP variation and relatively small address deltas between consecutive accesses. Conversely, the second phase features shorter latencies between memory accesses, fewer unique IPs, and significantly larger and wider-ranging address deltas. These phases alternate irregularly, either intermittently or repeatedly, posing challenges for traditional prediction tables. Furthermore, when a single IP is associated with multiple distinct access streams, traditional unified-table designs often prematurely evict infrequent yet important entries, particularly if a count-based confidence mechanism is employed. Similar patterns are also observed in other memory-intensive SPEC workloads, emphasizing the need for a prefetcher that quickly adapts to shifting access patterns and avoids prematurely losing critical pattern information.

Self-Attention. The self-attention mechanism imposes a significant memory demand due to the quadratic growth in memory complexity with increasing sequence length [5]. As LLMs evolve, running them locally on CPUs has become an attractive option for privacy and accessibility, particularly for lightweight tasks where compact models are practical. However, attention computations involve frequent data movement between on-chip caches and off-chip DRAM, exacerbating memory bottlenecks and limiting overall performance. We analyze memory access patterns of self-attention computations in Transformer models executed in PyTorch, where key operations (e.g., matrix multiplications) are dispatched to backend routines provided by Intel MKL [21]. Based on this analysis, we present two key observations that guide the design and optimization of a hardware prefetcher to accelerate memory accesses for attention-based workloads on CPUs.

Observation 1. *Self-attention in LLMs exhibits very high MPKI rates across all levels of the cache hierarchy.*

The quadratic memory complexity of the self-attention mechanism has a significant impact on the memory hierarchy behavior of LLMs. Figure 1 presents the Misses Per Kilo Instruction (MPKI) rates observed for self-attention workloads across various models with different sequence lengths in a single-core configuration (see Section IV for details). We observe that self-

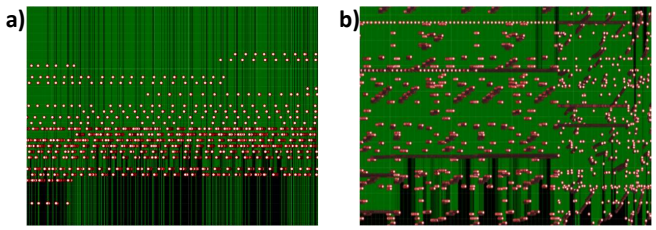

Fig. 2: Various access patterns in self-attention workloads exhibiting temporal evolution: (a) locally-regular pattern and (b) globally-irregular and unpredictable patterns.

attention workloads in LLMs incur a high number of cache misses at all levels of the hierarchy. The average MPKI values for the L1 data cache (L1D), L2 cache (L2C), and last-level cache (LLC) are 36.2, 11.6 and 10.9, respectively. Moreover, we observe that 94.0% of accesses that miss in the L2 also miss in the LLC, ultimately leading to costly DRAM accesses and highlighting significant inefficiencies in cache utilization. This underscores the importance of optimizing cache performance (especially at the L2) to reduce memory traffic and mitigate memory bottlenecks.

Observation 2. *Self-attention in LLMs exhibits a diverse and dynamic behavior in memory access patterns.*

We analyze the memory traces of the self-attention mechanism in LLMs and observe varying memory behaviors. In Figure 2, the x-axis represents the temporal order of memory accesses, while the y-axis corresponds to the accessed (red dot) memory addresses. A thin green line connects given accesses consecutively and visualizes the temporal flow of memory references.

Figure 2(a) presents a region illustrating dense clusters of accesses with high spatial locality and small, varying strides. These behaviors are typically associated with token-level operations such as embedding lookups. Figure 2(b) provides a zoomed-out view of the accesses shown in Figure 2(a), highlighting a phase spanning tens of thousands of addresses. Figure 2(b) exhibits phases similar to those in *mcf*, characterized by highly fragmented and irregular memory access patterns, which pose significant challenges for traditional prefetching strategies. In such cases, blind or overly aggressive prefetching can increase memory traffic and contention, ultimately degrading performance. These observations highlight the importance of an adaptive prefetcher that tunes its decisions and insertion destinations at runtime to maintain efficiency without overwhelming the memory hierarchy.

III. Zion: A Comprehensive and Adaptive Prefetcher

We propose Zion, a comprehensive, adaptive and lightweight prefetcher designed to accelerate memory-intensive workloads. At the core, it features ITSM, an independent temporal-spatial modules design that provides predictions by achieving comprehensive coverage of both temporal and spatial locality accesses. Additionally, Zion employs runtime feedback to dynamically

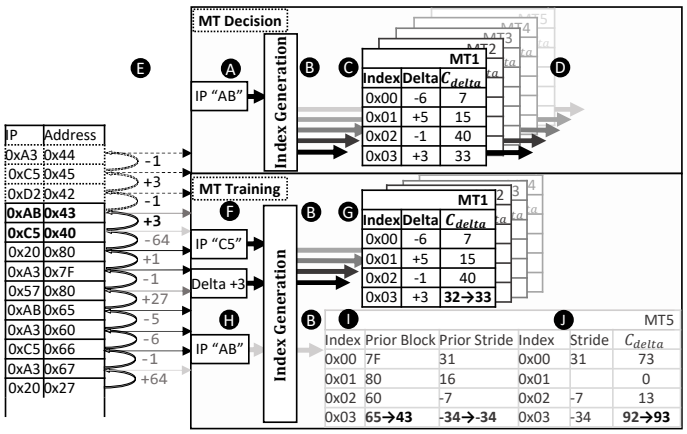

Fig. 3: Overview of ITSM.

adjust prefetch decisions and insertion destinations, optimizing performance while minimizing memory pressure.

A. ITSM: Independent Temporal-Spatial Modules

Independent Temporal-Spatial Modules (ITSM) consist of multiple prediction modules that leverage IP to predict memory access patterns by tracking both deltas and strides. Traditional prefetchers typically rely on a unified prediction table that globally tracks memory access transitions. Such a monolithic structure is prone to entry conflicts across unrelated data streams. Specifically, when count-based confidence is used, locally frequent but globally rare patterns often result in overly short-sighted predictions and poor responsiveness. Additionally, the unified table structure enforces coarse-grained prefetch management, which can penalize highly accurate predictions in otherwise effective local regions. Furthermore, accessing a large unified table suffers from scalability and responsiveness issues due to significant storage and lookup overhead. ITSM addresses these limitations by employing five monitoring-tables (MTs), four of which are each dedicated to a distinct delta range, and one MT specifically focuses on spatial accesses within a unified stride range.

Predictions. Figure 3 presents a high-level overview of ITSM. To make prefetching decisions, the number of candidates issued by Zion is limited to those targeting addresses within the same physical page, since prediction accuracy typically declines when delta or stride magnitudes increase and cross page boundaries [22]. The workload begins by processing each demand request and extracting the IP associated with the corresponding memory access (Ⓐ). We then compress the IP using a hash function [23] and simultaneously index the five MTs (Ⓑ). Four MTs use deltas (the difference between the current and previous addresses) to capture temporal locality, while a single MT targets strides (the difference between two consecutive blocks accessed by the current IP [24]) to capture spatial locality, enabling comprehensive pattern recognition. ITSM distributes the prediction process across these five distinct MTs (Ⓒ), each predicting the most likely next delta or stride for a given IP to generate prediction addresses within the Zion prefetcher (Ⓓ).

Training. Besides making prediction decisions, ITSM also leverages every demand access to dynamically update the MTs, ensuring that they remain relevant to changing memory access patterns. For the first four MTs (which handle deltas), the training phase begins by processing each memory address to compute the delta (E). This computed delta, based on the range, determines which MT should be updated. Similar to the prediction phase, ITSM uses the IP (F), to index the appropriate MT (B) and update it. If a matching entry is found in the MT, the confidence score of the corresponding entry is incremented. Otherwise, a new entry is inserted into the corresponding MT (G). The training phase ensures that the prediction model continuously adapts to evolving memory access patterns, and table segmentation mitigates conflicts between low- and high-delta streams that share the same IP.

Simultaneously, the fifth MT targets spatial locality by predicting strides. During the training process, currently accessed IP (H) is compressed [23], to index the corresponding table (B). Before updating the stored prior block of an IP to the current block (I), using these two subsequently accessed blocks, the current stride is computed and compared with the prior stride stored. On a mismatch, the stored stride is updated; otherwise, the confidence score is incremented for the matching entry (J). This improves coverage of alternating strides [7] by storing multiple variants of consecutively repeated strides for the same IP.

Range-Segmented Monitoring-Tables. To ensure efficient hardware realization for predictions, ITSM adopts a cache-like structure for each segmented MT. Each MT is indexed using the encoded IP [23] and retrieves the corresponding delta or stride along with a 16-bit confidence score, C_{delta} (saturated at 65535). Each MT uses a Least Recently Used (LRU) replacement policy, allowing to adapt to evolving memory behavior while remaining efficient and scalable for modern memory systems. Each delta-based MT is independently sized to reflect the observed frequency and reuse characteristics of deltas within its range. Based on our analysis, smaller deltas occur more frequently and exhibit shorter reuse distances; therefore, larger tables are allocated to accommodate more entries. Conversely, larger deltas, which are less frequent and have longer reuse distances, thus smaller tables are assigned to reduce storage overhead. This design effectively mitigates count-based confidence bias, caused by the dominance of frequent deltas, while balancing storage overhead with prediction performance.

To cover deltas in the range $[-63, +63]$ for a system with a standard 4KB page size and 64B cacheline, we used a software-like logarithmic partitioning to define the delta ranges for each MT as follows:

- **MT1**: $[-8, 0) \cup (0, +8]$ with 512 entries, covering the smallest and most frequent deltas, which typically indicate accesses with high spatial locality.
- **MT2**: $[-16, -8] \cup [+8, +16]$ with 320 entries, handling slightly larger deltas while maintaining separation from the smaller range.
- **MT3**: $[-32, -16] \cup [+16, +32]$ with 128 entries, capturing moderately infrequent patterns.

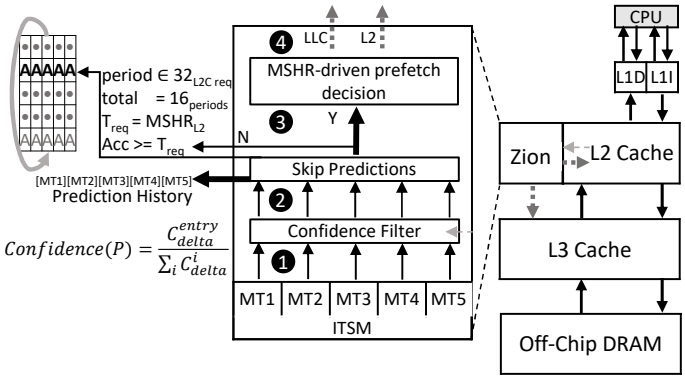

Fig. 4: Zion architecture in the memory hierarchy.

- **MT4**: $[-63, -32] \cup [+32, +63]$ with 64 entries, targeting the largest and least frequent deltas.

Beyond delta-based prediction, we implement a stride-based MT, to keep track of multiple consecutively repeated strides per IP:

- **MT5**: $[-64, 0) \cup (0, +64]$ with 320 entries, focusing on strides for spatial locality.

We relax the strict non-overlap constraint at boundary values (e.g., allowing shared deltas such as $\pm 8, \pm 16, \pm 32$), which improves prediction accuracy at delta range boundaries.

B. Dynamic Adjustment of Prefetcher Behavior

Although Zion provides both temporal and spatial pattern recognition, unmanaged and overly aggressive predictions may cause severe cache pollution, increased memory pressure, and degraded performance [12], [25]. To address this, Zion employs three adjustment strategies to dynamically manage the behavior of ITSM at the L2 cache level, as shown in Figure 4.

For every MT prediction (1), a confidence-based filtering mechanism applies empirically selected thresholds of 20%, 25%, 30%, and 40% for delta-based MT1 to MT4, and 1% is set for stride-based MT5 (2). The confidence is computed as the ratio between the candidate's confidence score and the total number of updates corresponding to a given IP in the MT. Candidates that do not meet the assigned threshold are discarded, minimizing inaccurate prefetches and cache pollution.

Accuracy Monitoring. Zion prefetcher evaluates each MT's performance over 512 accesses by using a 32 access period effectiveness level. If MT's level falls below the required threshold, determined by hardware feedback, it is temporarily disabled.

For each MT, confidence-filtered candidates are recorded in the prediction history table (3) using the First-In-First-Out (FIFO) replacement policy. Each entry stores a 6-bit page-relative predicted block number and a 10-bit encoded page number. On every L2 hit request, the access is encoded and searched across all prediction history tables. If an MT had predicted a given block, a 5-bit counter is incremented. Every 32-access period, the numbers of successful MT predictions are encoded in three-level effectiveness (using a 2^n scaling)

TABLE 1: Storage overhead of Zion

Structure	Field (#bits in each entry)	Entries	#Bits
ITSM	MT1: IP(16)+Delta(3)+C_{delta}(16)	512	17920
	MT2: IP(16)+Delta(3)+C_{delta}(16)	320	11200
	MT3: IP(16)+Delta(4)+C_{delta}(16)	128	4608
	MT4: IP(16)+Delta(5)+C_{delta}(16)	64	2368
	MT5: IP(16)+Delta(6)+C_{delta}(16)	320	12160
	Previous: Stride(6)+Block(58)	320	20480
LRU	MT1(9)+MT2(9)+MT3(7)+MT4(6)+MT5(9)		40
	ITSM Total:		**8.4KB**
Predictions	#MT(5) x (Page(10)+Page Block(6))	64	5120
Period	Global Counter(5) | Effectiveness(2)	16	37
Metadata	Counter(5)+FIFO(4)	5	45
	Tracker Total: **0.64KB**	Zion Total: **9.03KB**	

using 2-bits in the 16-entry period table before advancing to the next period. After 15 periods, based on effectiveness, MT is temporarily disabled (❸).

Real-time System Feedback. The required effectiveness level is determined by the availability of L2 Miss Status Holding Register (MSHR). When 25% or 12.5% of MSHR is available, an MT, must have the second or third level, respectively, to remain active. To avoid false-positive disabling, Zion continuously monitors inactive MTs, and if a new, in-progress, period monitoring reaches the required threshold, MT is reactivated. Moreover, based on MSHR, the L2 pressure is alleviated when occupancy exceeds thresholds (75% spatial, 50% temporal locality MTs), Zion redirects prefetch requests to LLC cache, which maintains overall performance (❹).

C. Storage Overhead

Table 1 summarizes the storage overhead of Zion. The primary storage overhead arises from maintaining the five modules in ITSM, along with the LRU policy used to manage their respective tables. A hash function is used to encode the IP into 16 bits for indexing all prediction tables. Delta values are encoded using N-bit representations, with the number of bits tailored to each segmented delta range. The confidence score C_{delta} is allocated 16 bits, to track a broad range of observed repetitions and to ensure robust prediction accuracy. In addition, Zion uses a lightweight accuracy monitoring, using a prediction table and accuracy table requiring only 128 bytes per module. Overall, Zion is lightweight, requiring only 9.03KB of storage for the 1-core configuration (see Table 2). This achieves 12.6× and 2.2× lower storage overhead than SOTA prefetchers Bingo [9] and Berti [8].

IV. Methodology

Simulation Setup. We evaluate the performance of Zion using the ChampSim simulator [26], which has been employed in the 2nd and 3rd Data Prefetching Championships [27], [28]. We simulate the Intel Alder Lake (GLC) [29] multi-core processor as the default microarchitecture, supporting configurations ranging from 1-core to 8-core systems. Table 2(A) outlines the system parameters. To explore architectural diversity, we also simulate the Apple A14 Firestorm processor, representing modern mobile and edge platforms, in a single-core configuration. Its detailed parameters are listed in Table 2(B).

TABLE 2: Simulated system parameters

(A) (Default) Intel Alder Lake (GLC)	
Core	1–8 cores, 6-wide fetch/execute/commit, 512-entry ROB, 128/72-entry LQ/SQ, perceptron branch predictor
L1/L2 Cache	Private, 48KB/1.25MB, 64B line, 12/20-way, 16/48 MSHRs, LRU, 5/15-cycle round-trip latency
LLC	3MB/core, 64B line, 12-way, 64 MSHRs/slice, LRU, 55-cycle round-trip latency
(B) Apple A14 Firestorm (A14)	
Core	1 core, 8-wide fetch/execute/commit, 632-entry ROB, 148/106-entry LQ/SQ, perceptron branch predictor
L1/L2 Cache	Private, 128KB/4MB, 64B line, 8/16-way, 16/48 MSHRs, LRU, 3/16-cycle round-trip latency
LLC	Shared, 8MB, 64B line, 16-way, 64 MSHRs/core, LRU, 24-cycle round-trip latency
Main Memory (Both Architectures)	
GLC/A14	**1C**: 1 channel, 1 rank per channel;
GLC	**2C**: 2 channels, 2 ranks per channel; **4C–8C**: 4 channels, 2 ranks per channel;
Both	8 banks per rank, DDR4-3200 MTPS, 64b data-bus per channel, 2KB row buffer, tRCD/tRP/tCAS=12.5ns

TABLE 3: Evaluated workloads

Suite	Workload
SPEC 2006	gcc, gobmk, libquantum, lbm, astar, wrf sphinx3, xalancbmk
SPEC 2017	perlbench, gcc, bwaves, mcf, cactuBSSN, lbm wrf, xalancbmk, x264, fotonik3d, roms
Self-Attention	Pythia: 256, 512, 1024 2048, 4096 (Tokens) GPT: 256, 512, 1024 2048, 4096 (Tokens) OPT: 256, 512, 1024 2048, 4096 (Tokens)

Evaluated Workloads. We evaluate Zion prefetcher on a set of memory-intensive workloads from SPEC CPU2006 [17] and SPEC CPU2017 [18], to assess its effectiveness for general-purpose CPU workloads. We select 37 traces, each exhibiting LLC miss per kilo instructions (MPKI) greater than 1 in the baseline system without prefetching. These evaluations include both homogeneous configurations (running identical copies of the same trace) and heterogeneous configurations (running distinct traces, one per core). For every benchmark cache is initialized through warm-up of 50M sim-point instructions and statistics are kept for the next 200M sim-point instructions. Table 3 summarizes the workloads evaluated in this study.

We also evaluate Zion using three widely adopted open-source language model variants: Pythia [30], GPT [31], and OPT [32], which represent compact LLM architectures optimized for edge and CPU-based inference [19], across various sequence lengths. Models are decoder-only transformers [33], pre-trained with causal modeling, executed in PyTorch with CPU-optimized backend libraries [20]. To collect self-attention traces from the models, we use the Intel Pin dynamic binary instrumentation tool [34]. For all simulations, the first 20% of instructions are used for warm-up, and the remaining 80% are used for evaluation.

Compared Prefetchers. We compare Zion against four SOTA hardware prefetchers: IPCP [7], Bingo [9], SPP [6], and Berti [8]. For comparison, we employ a simple next-line prefetcher at the L1D cache, while each evaluated prefetcher is deployed at the L2 cache. We report normalized weighted speedup [7], [35] over a no-prefetch baseline.

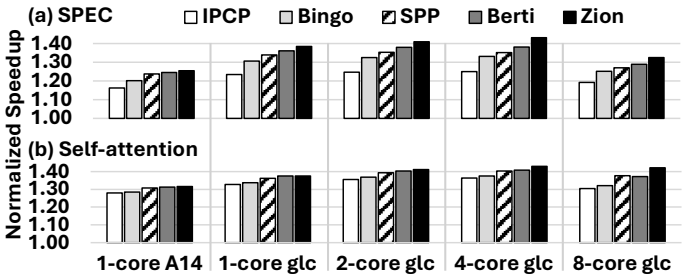

Fig. 5: Performance improvements of prefetchers on (a) SPEC 2006, SPEC 2017 and (b) self-attention workloads across varying core counts.

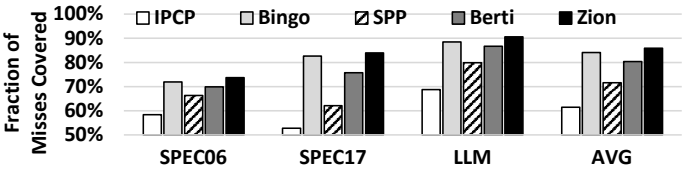

Fig. 6: Coverage of LLC cache misses achieved by various prefetchers in a 4-core configuration.

V. Experimental Results

A. Performance of SPEC workloads

Figure 5(a) presents the normalized speedup of all evaluated prefetchers running 37 homogeneous SPEC workloads under various system configurations, including 1-core (Apple A14 and GLC), as well as multi-core (2/4/8-core GLC). Across all configurations, Zion consistently outperforms all SOTA designs, achieves speedup of 25.5%, 38.5%, 40.9%, 43.2% and 32.5% over the baseline, respectively. In particular, under the optimal 4-core configuration, Zion surpasses IPCP, Bingo, SPP, and Berti by 18.2%, 10.1%, 8.0%, and 5.1%, respectively. Moreover, the memory subsystem saturation, at a higher core count degrades the performance at 8-core for all prefetchers. At the other end, on 1-core A14, the 6.4× smaller L2 cache capacity limits the gains of SPP, Berti, and Zion, resulting in a small performance gap.

B. Performance of Self-Attention workloads

Figure 5(b) illustrates the geometric mean speedup across various prefetchers on homogeneous self-attention workloads. Zion achieves 31.7%, 37.6%, 41.3%, 43.0%, and 42.2% speedup on A14 (1-core) and GLC (1-core, 2-core, 4-core, and 8-core). These results demonstrate that LLMs with equivalent internal architectures achieve consistent speedups in self-attention mechanisms. Notably, in all multi-core configurations, Zion outperforms other prefetchers in every workload. As the core count increases, pattern interference increases and prefetcher adaptability becomes increasingly important. As a result, Zion in the 8-core configuration outperforms IPCP, Bingo, SPP, and Berti by 11.8%, 10.0%, 4.4%, and 5.1%, respectively. Notably, the marginal improvement over the strongest competing prefetcher in any given workload is between 1.54% and 5.9%.

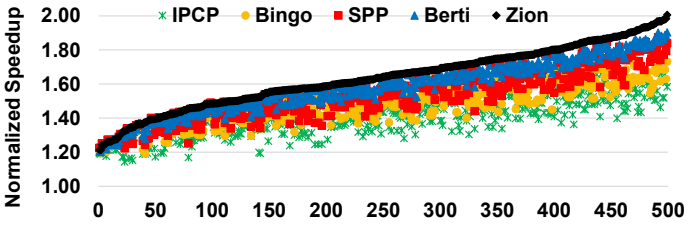

Fig. 7: Weighted speedup of prefetchers on heterogeneous SPEC workload mixes in a 4-core system.

C. Homogeneous Pattern Coverage

We evaluate prefetch coverage, defined as the ratio of cache misses eliminated by the prefetcher to the number of misses in the baseline (no prefetching). Each prefetcher is trained using L1-cache misses, while predicted lines are inserted into the L2 and LLC. In the 4-core homogeneous configuration, Figure 6 shows that Zion achieves LLC coverage rates of 73.6%, 84.0%, and 90.5% on SPEC 2006, SPEC 2017, and self-attention workloads, respectively. The next best prefetcher, Berti, achieves coverage rates of 69.8%, 75.8%, and 86.6% on the same workloads.

D. Heterogeneous Workloads in 4-Core Systems

To comprehensively evaluate the performance, we generate 500 heterogeneous 4-core mixes by randomly selecting traces from the evaluated SPEC and self-attention benchmarks, assigning a distinct trace to each core. Figure 7 shows the weighted speedup over the no-prefetch baseline, with mixes sorted in ascending order of Zion's performance. Overall, Zion achieves an average speedup of 63.8% over the baseline, outperforming IPCP, Bingo, SPP, and Berti by 22.3%, 15.0%, 9.9%, and 5.5%, respectively.

VI. Conclusion

This paper presents a Zion prefetcher, a comprehensive and adaptive hardware prefetcher. By integrating ITSM with five specialized prediction modules, Zion achieves comprehensive coverage across diverse memory access patterns. Additionally, Zion dynamically tunes prefetching behavior based on runtime feedback to mitigate memory pressure and enhance performance, while remaining lightweight. Evaluations show that Zion consistently outperforms state-of-the-art prefetchers across both SPEC and self-attention workloads. Overall, Zion provides a practical and extensible foundation for future CPU-based systems targeting efficient memory-intensive computing and on-device LLM inference.

Acknowledgements

We thank the anonymous reviewers for their constructive and insightful feedback. This research is supported in part by the National Science Foundation under Grants CNS-2310422, CNS-2152497, CNS-2431516 and NSF 2450524. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

REFERENCES

[1] W. A. Wulf et al., "Hitting the memory wall: Implications of the obvious," ACM SIGARCH computer architecture news, vol. 23, no. 1, pp. 20–24, 1995.

[2] X.-H. Sun and X. Lu, "The memory-bounded speedup model and its impacts in computing," Journal of Computer Science and Technology, vol. 38, no. 1, pp. 64–79, 2023.

[3] T. Zhang et al., "Nomad-attention: Efficient llm inference on cpus through multiply-add-free attention," arXiv preprint arXiv:2403.01273, 2024.

[4] H. Shen et al., "Efficient llm inference on cpus," arXiv preprint arXiv:2311.00502, 2023.

[5] T. Dao et al., "Flashattention: Fast and memory-efficient exact attention with io-awareness," NIPS, pp. 16 344–16 359, 2022.

[6] J. Kim et al., "Path confidence based lookahead prefetching," in MICRO, 2016, pp. 1–12.

[7] S. Pakalapati et al., "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in ISCA, 2020, pp. 118–131.

[8] Navarro-Torres et al., "Berti: an accurate local-delta data prefetcher," in MICRO, 2022, pp. 975–991.

[9] M. Bakhshalipour et al., "Bingo spatial data prefetcher," in HPCA, 2019, pp. 399–411.

[10] P. Zhang et al., "Resemble: reinforced ensemble framework for data prefetching," in SC22, 2022, pp. 1–14.

[11] D. Joseph et al., "Prefetching using markov predictors," in ISCA, 1997, pp. 252–263.

[12] X. Lu et al., "Apac: An accurate and adaptive prefetch framework with concurrent memory access analysis," in ICCD, 2020, pp. 222–229.

[13] Z. Shi et al., "A hierarchical neural model of data prefetching," in ASPLOS, 2021, pp. 861–873.

[14] K. J. Nesbit et al., "Data cache prefetching using a global history buffer," in HPCA, 2004, pp. 96–96.

[15] S. Ainsworth et al., "Graph prefetching using data structure knowledge," in ICS, 2016, pp. 1–11.

[16] A. Basak et al., "Analysis and optimization of the memory hierarchy for graph processing workloads," in HPCA, 2019, pp. 373–386.

[17] "SPEC CPU2006 Benchmark Suite," http://www.spec.org/cpu2006/, 2006.

[18] "SPEC CPU2017 Benchmark Suite," http://www.spec.org/cpu2017/, 2017.

[19] Z. Liu et al., "Mobilellm: Optimizing sub-billion parameter language models for on-device use cases," arXiv preprint arXiv:2402.14905, 2024.

[20] A. Paszke, "Pytorch: An imperative style, high-performance deep learning library," arXiv preprint arXiv:1912.01703, 2019.

[21] E. Wang et al., "Intel math kernel library," HPC on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures, pp. 167–188, 2014.

[22] S. Somogyi et al., "Spatio-temporal memory streaming," ACM SIGARCH Computer Architecture News, vol. 37, no. 3, pp. 69–80, 2009.

[23] Navarro-Torres et al., "A complexity-effective local delta prefetcher," IEEE Transactions on Computers, 2025.

[24] J. W. Fu et al., "Stride directed prefetching in scalar processors," MICRO, vol. 23, no. 1-2, pp. 102–110, 1992.

[25] S. Srinath et al., "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in HPCA. IEEE, 2007, pp. 63–74.

[26] N. Gober et al., "The Championship Simulator: Architectural Simulation for Education and Competition," 2022.

[27] "2nd cache replacement championship." https://crc2.ece.tamu.edu/, 2017.

[28] "3rd cache replacement championship." https://crc2.ece.tamu.edu/, 2019.

[29] E. Rotem et al., "Intel alder lake cpu architectures," MICRO, pp. 13–19, 2022.

[30] S. Biderman et al., "Pythia: A suite for analyzing large language models across training and scaling," in ICML, 2023, pp. 2397–2430.

[31] A. Radford et al., "Language models are unsupervised multitask learners," OpenAI blog, p. 9, 2019.

[32] S. Zhang et al., "Opt: Open pre-trained transformer language models," arXiv preprint arXiv:2205.01068, 2022.

[33] A. Radford, "Improving language understanding by generative pre-training," 2018.

[34] V. J. Reddi et al., "Pin: a binary instrumentation tool for computer architecture research and education," in WCAE, 2004, pp. 22–es.

[35] X. Lu et al., "Chrome: Concurrency-aware holistic cache management framework with online reinforcement learning," in HPCA, 2024, pp. 1154–1167.