PF-LLM: Large Language Model Hinted Hardware Prefetching

Ceyu Xu*†
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong
eeentropy@ust.hk

Xiangfeng Sun*
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong
xsunbv@connect.ust.hk

Weihang Li
Duke University
Durham, USA
weihang.li@duke.edu

Chen Bai
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong
eecbai@ust.hk

Bangyan Wang
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong
wangbangyan@gmail.com

Mengming Li†
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong
mengming.li@connect.ust.hk

Zhiyao Xie
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong
eezhiyao@ust.hk

Yuan Xie
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong
yuanxie@ust.hk

Abstract

Hardware data prefetching is a critical technique for mitigating memory latency in modern processors. While sophisticated hardware prefetching algorithms exist, their exclusive reliance on runtime information limits their ability to adapt quickly and comprehend broader program context. Our key insight is that the optimal prefetching strategy for a load instruction is often discernible from its static code context—a task at which experienced developers excel. This motivates our central question: can a Large Language Model (LLM) be trained to perform this analysis automatically?

We introduce **PF-LLM**, an LLM fine-tuned to analyze the assembly context surrounding a load instruction and generate prefetching hints. These offline-generated hints are consumed at runtime by **LMHint Prefetcher**, a lightweight hardware prefetcher ensemble designed to leverage this static guidance.

Our approach boosts the performance of the on-chip hardware prefetcher by moving the hard "when, how, and how aggressively to prefetch" decisions **out of the runtime hardware** and into an **offline LLM-powered analysis**. This turns the on-chip prefetcher into a **zero-latency, oracle-level system** that always follows the best prefetching policy for every single load instruction. Our evaluation shows that our approach achieves a **9.8%** instruction-per-cycle (IPC) improvement on average for memory-intensive SPEC 2017 benchmarks over state-of-the-art hardware prefetching baselines and **18.9%** improvement on average over state-of-the-art ensemble methods, demonstrating the significant potential of leveraging LLMs to guide microarchitectural decisions.

CCS Concepts: • **Computer systems organization → Architectures**; *Serial architectures.*

Keywords: Prefetching, Hardware Prefetcher, Large Language Model, CPU Microarchitecture

ACM Reference Format:
Ceyu Xu, Xiangfeng Sun, Weihang Li, Chen Bai, Bangyan Wang, Mengming Li, Zhiyao Xie, and Yuan Xie. 2026. PF-LLM: Large Language Model Hinted Hardware Prefetching. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779212.3790202

*Both authors contributed equally to this research.
†Corresponding author.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790202

1 Introduction

Advancing single-core processor performance is an enduring challenge in computer architecture. Despite the rise of multi-core systems, GPUs, and other accelerators that exploit massive parallelism, Amdahl's Law underscores the persistent importance of single-thread speed for reducing

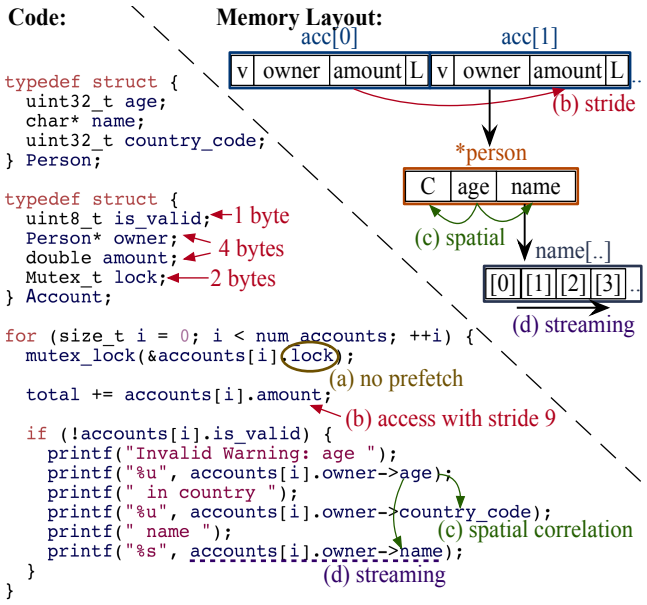

Figure 1. An example of a bank account management C code demonstrating easily identifiable memory access patterns. The patterns include (a) no prefetching for atomic operations, (b) strided prefetching for struct array traversals, (c) spatial prefetching for struct member accesses, and (d) streaming prefetching for sequential string reads.

application latency. With gains from instruction-level parallelism (ILP) plateauing, the most significant obstacle remains the "memory wall." This term describes the gap between processor speed and memory latency, where a single DRAM access typically takes hundreds of cycles to finish in modern processors. To breach this wall, speculative techniques, especially data prefetching [3, 11, 25], have become a primary focus of modern processor design to hide memory latency.

Data prefetching speculatively moves data to higher levels of the memory hierarchy before the core demands it. The effectiveness of a hardware prefetcher is rooted in its ability to recognize and act upon specific memory access patterns. Extensive research has developed specialized prefetchers targeting specific patterns such as streams [29], strides [17, 29, 32], spatial localities [5, 7, 30, 33], and irregular sequences [2, 18, 21, 35**?**]. This specialization, however, leads to a dilemma. Real-world applications are rarely monolithic, instead comprising diverse and dynamically shifting execution phases, each with its own dominant memory behavior. A single specialized prefetcher is therefore inherently inadequate for covering all phases of a workload. To achieve robust, comprehensive coverage, recent designs have been focusing on combining multiple specialists into an *ensemble* architecture [13, 19, 22, 27]. This, however, shifts the critical challenge from designing individual prefetchers to dynamically orchestrating the ensemble.

When using a prefetcher ensemble, the performance of the entire system becomes contingent on a selection policy, which must correctly match dynamic program phases with the optimal sub-prefetcher selection. This is because an ill-suited selection degrades performance by evicting useful cache data and wasting memory bandwidth. Prior ensemble methods rely on online learning to derive this policy at runtime. However, these online methods face two major challenges. On one hand, they require a slow and costly trial-and-error convergence period to learn effective policies, which limits their ability to adapt to rapid phase changes. On the other hand, strict on-chip area and latency constraints restrict these mechanisms to simple heuristics, preventing them from leveraging a wider program context to make more informed decisions.

Our key insight is that the static code context contains sufficient information to predict memory access patterns, allowing us to analyze the static code context to provide guidance for online prefetching decisions. To illustrate, Figure 1 presents a simple C code example where four distinct memory access patterns are readily identifiable. These include (a) Lock Acquisition, an atomic operation requiring no prefetching; (b) Strided Access, a predictable, fixed-stride pattern from a loop; (c) Spatial Access, where an access to a struct is followed by accesses to other fields; and (d) Streaming Access, a classic sequential read of a string. This example highlights that even a simple program requires a mix of prefetching strategies. Crucially, it also shows that the appropriate strategy can often be inferred directly from the static code structure.

This observation motivates our central question: **If an experienced developer can identify these memory access patterns from code, can a modern large language model (LLM) perform the same task automatically?** We propose to fine-tune an LLM, which we name **PF-LLM**, to predict the optimal prefetching strategy for individual load instructions based on their assembly code context. To achieve this, PF-LLM is trained on a large dataset of assembly code and its corresponding ground-truth prefetching strategies, obtained through architectural simulation. Our approach leverages the large compute budget of offline analysis to enable a much deeper static code inspection than is feasible at runtime. For each load instruction, the trained PF-LLM model outputs three predictive hints: (1) the prefetcher selection (e.g., stride or stream), (2) the prefetch degree, and (3) a demand request filter to improve the accuracy of certain sub-prefetchers. At runtime, a lightweight hardware prefetcher ensemble, which we name **LMHint Prefetcher**, utilizes these hints to guide its decisions. By using these offline-generated hints, LMHint Prefetcher avoids the costly online training period of conventional ensemble methods, allowing it to adapt more effectively to diverse workloads.

Overall, we make the following contributions:

1. We propose and fine-tune PF-LLM, an LLM that analyzes the assembly context of a load instruction to predict its optimal prefetching strategy.
2. We introduce LMHint Prefetcher, a lightweight prefetcher ensemble designed to consume the offline-generated hints from PF-LLM to make effective online prefetching decisions.
3. Our evaluation shows that our approach significantly improves performance, achieving a **18.9%** IPC improvement over state-of-the-art ensemble methods and a **9.8%** improvement over a state-of-the-art single prefetcher on memory-intensive SPEC2017 benchmarks.
4. To our knowledge, this is the first work to leverage an LLM's code understanding capabilities to provide hints for a dynamic microarchitectural mechanism.

2 Background

2.1 Prefetcher Ensemble

A prefetcher ensemble is typically composed of multiple time-proven hardware prefetchers as *sub-prefetchers* as well as an *orchestration layer*. The orchestration layer manages the sub-prefetchers and make the whole ensemble expose the same interface like only one single prefetcher to the core.

A prefetcher's interface includes the demand requests as inputs (for training the prefetcher) and an output port where the prefetch requests are been issued from. When multiple, distinct prefetchers are combined into an ensemble, a dedicated orchestration layer is required to manage their interactions and resource contention. As illustrated by component (c) in Figure 2, this orchestration layer is responsible for making two fundamental decisions: 1) When a demand request arrives, which one or more sub-prefetchers should this demand request be routed to as shown in Figure 2(a)? 2) When one or more sub-prefetchers issue prefetch requests, which request or requests should be selected to issue to the next level of the memory hierarchy as shown in Figure 2(b)? Existing approaches have focused on either side of the problem [13, 19] or both sides of the problem [13] at once.

However, one significant limitation we found of existing approaches is their poor adaptability to sophisticated prefetchers. When applied to sophisticated modern prefetchers like Pythia [6], existing orchestration algorithms [19, 22] can disrupt the internal state and interfere with complex pattern detection of advanced sub-prefetchers, leading to significant performance degradation. Effectively managing an ensemble of advanced prefetchers requires more fine-grained, context-aware control over the inputs and outputs of each component. We will demonstrate this phenomenon in Section 6.

In our work, we leverage the prefetching degree as a key control parameter in the orchestration layer to manage prefetching aggressiveness. This control, combined with other mechanisms discussed in Section 4, is essential for creating an effective ensemble that can leverage the power of both simple and sophisticated prefetchers.

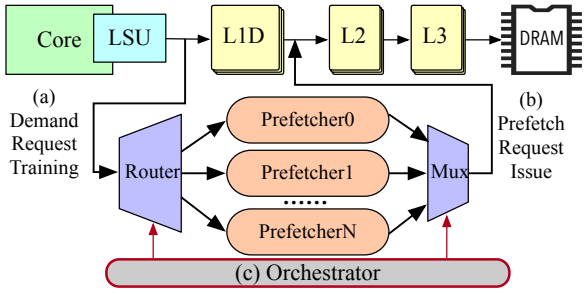

Figure 2. Orchestration Layer for Prefetcher Ensemble

2.2 Offline vs. Online Prefetching

Hardware prefetchers face complexity constraints. In modern CPU cores that operate at frequencies as high as 5GHz, the time budget for a prefetching decision is often at the sub-nanosecond level. This latency requirement, along with tight on-core area and power constraints, restricts online prefetching to simple algorithms with cheap logic. Consequently, these algorithms cannot afford to analyze a longer program context or make more informed decisions.

Offline prefetchers are therefore explored as a way to leverage a larger compute budget to make more precise prefetch decisions. Two primary methods that exploit this budget are compiler-based approaches and profile-guided approaches.

Compiler-based approaches [9] rely on manually designed heuristics to determine where to place prefetch instructions at compile time. While this approach shows improvement over the baseline, its effectiveness is dependent on human experts to design the heuristics. As usually these heuristics are triggered by a specific code pattern, it lacks the ability to reason about the prefetching decisions at a program level and cannot adapt to more complex code structures.

The profile-guided optimization (PGO) approaches [36**?**], on the other hand, compile and profile a program one or more times to gather "what-if" information about prefetching decisions until an effective policy is found. Although PGO shows great performance across a wide range of programs, its key problem is that it requires the program's input data to collect the correct profile information, which is not always available. When a program's input is unknown or varies greatly, PGO is inapplicable.

Furthermore, both compiler-based and profile-guided approaches rely on the recompilation of a program to inject prefetching instructions or hints. If an application is closed-source and only a binary is available, both types of offline prefetch analysis are inapplicable.

3 Why an LLM is a Good Choice for Prefetching?

Existing prefetching techniques face a difficult trade-off. Hardware prefetchers are constrained by the complexity, while offline compilers and profile-guided methods suffer from brittle heuristics or dependencies on source code and specific input data. We argue that a Large Language Model (LLM) fine-tuned for prefetching can uniquely overcome these limitations by combining deep offline analysis with the adaptability of a learned approach.

Deep Code Comprehension

A primary advantage of an LLM is its ability to reason about code at a higher level of abstraction than traditional methods. Hardware prefetchers typically identify patterns from streams of memory addresses, limiting them to temporal regularities like fixed strides. Similarly, compiler heuristics often rely on local loop analysis or pattern matching. In contrast, an LLM can infer the underlying semantics from the code context. For example, it can distinguish between traversing a simple array, an array of structs, or a linked list, each of which implies a different optimal prefetching strategy. Furthermore, the large context windows of modern LLMs enable a higher-level analysis of program structure that is infeasible to perform in compilers or hardware prefetchers.

Automated Heuristic Discovery

Our approach uses an LLM to automatically learn effective prefetching heuristics from data. This stands in sharp contrast to compiler-based techniques that depend on hand-crafted heuristics. Manually designing these rules is a labor-intensive process that requires significant domain expertise and often results in policies that are brittle and difficult to generalize. A learned approach, however, can discover more complex and nuanced patterns from a vast dataset of code. Furthermore, unlike static hand-crafted rules, which could become outdated after the release of newer generation hardware, learned heuristics can adapt to new programming idioms, new hardware features, and new workload characteristics by fine-tuning the model on new training data. This bypasses both the rigidity of compiler heuristics and the costly online training required by hardware ensemble prefetchers.

Generalization from Collective Experience

Profile Guided Optimization (PGO) learns from experience, using execution profiles to test several "what-ifs" to make offline prefetching decisions. However, its knowledge is specific to the application being profiled and is often sensitive to the particular input data used during the profiling runs. Our approach represents a conceptual evolution of this idea. The LLM is trained on a diverse corpus representing aggregate experience from many different programs. Instead of learning the behavior of a single program, the LLM learns to generalize the relationship between static code patterns and dynamic memory behavior across a wide range of applications. Consequently, PF-LLM acts as a proxy for empirical analysis, capable of inferring an effective prefetching policy for a new, unseen program without the need for an application-specific profile. This obviates the dependency on input data and makes the optimization far more robust. Critically, PF-LLM operates on assembly code. This makes our technique independent of both the original source code and specific program inputs, extending its applicability to pre-compiled, closed-source binaries where profile-guided optimization and many compiler-based approaches are not viable.

Leveraging Pre-trained Foundation Models

The feasibility of our method is rooted in the ability to fine-tune powerful, pre-trained coding models. These models have already been trained on billions of lines of code [16], endowing them with a general understanding of programming languages and structures. By specializing such a model for prefetching, we leverage this vast prior knowledge, making our approach far more practical than training a model from scratch.

Ultimately, the efficacy of this approach is validated by its results. Our PF-LLM model achieves a remarkable 95% accuracy in correctly predicting the optimal prefetching policy from assembly context alone. This high accuracy translates directly into significant performance gains at runtime. The LMHint Prefetcher prefetcher, guided by these offline-generated hints, delivers an average IPC improvement of **18.9%** over state-of-the-art ensemble methods and **9.8%** over a highly optimized single prefetcher on memory-intensive benchmarks. This confirms that leveraging an LLM for offline analysis is a powerful and effective new direction for hardware prefetching.

4 PF-LLM Model and LMHint Prefetcher

This section details the design of our proposed framework, which consists of two primary components: the PF-LLM model for offline analysis and the LMHint Prefetcher hardware prefetcher for runtime execution. The overall workflow is illustrated in Figure 3. The rest of this section is organized as follows: We first describe the architecture of the PF-LLM model (Section 4.1), its training methodology (Section 4.2), and the offline hint generation process (Section 4.3). Subsequently, we detail the design of the LMHint Prefetcher hardware, which consumes these hints to guide its online prefetching decisions (Section 4.4). Implementation and experimental setup details are provided in Section 5.

Domain of PF-LLM and LMHint Prefetcher: It's worth pointing out beforehand that our work focuses on **data** prefetching for the **L1D** cache within a modern superscalar

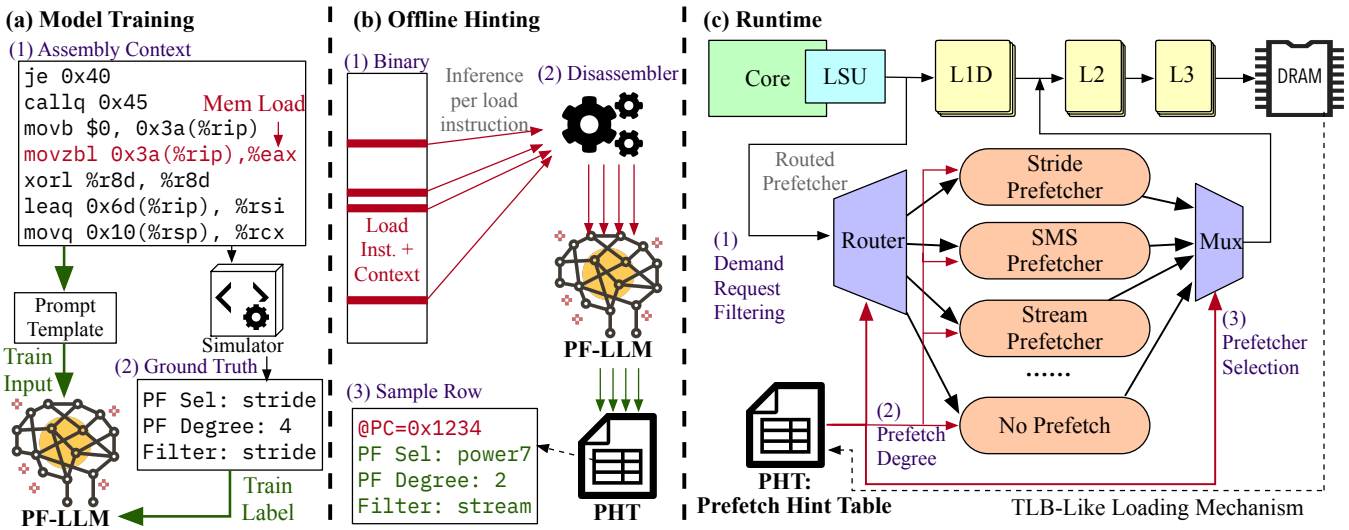

Figure 3. Overview of our approach. (a) The PF-LLM model is fine-tuned using ground-truth prefetching policies obtained from architectural simulation. (b) The trained PF-LLM model generates prefetching hints offline from an application's assembly code. (c) The LMHint Prefetcher hardware prefetcher consumes these hints at runtime to make informed online prefetching decisions.

processor core. For simplicity and clarity, our experiments and evaluations are confined to a single-core context, and we exclusively focus on the x86-64 instruction set architecture. We believe that demonstrating the efficacy of our approach at the microarchitectural level will motivate future research into its application in larger, multi-core systems.

4.1 PF-LLM Model Architecture

The PF-LLM model is designed to analyze static code context and produce hints for the runtime prefetcher. We now describe its architecture, input representation, and output format.

Model Architecture. To leverage the extensive knowledge embedded in existing foundation models, we fine-tune a pre-trained, coding-specialized LLM. Since prefetching is a highly specialized task, a smaller model offers sufficient capability while enabling faster training and requiring a smaller GPU memory footprint. After a review of available models, we selected **Qwen-2.5-Coder-0.5B-Instruct** [16], which exhibits state-of-the-art coding capabilities at its size and best fits our requirements. Our experiments fine-tune this model on our prefetching dataset without altering its underlying model architecture.

Input. The PF-LLM model accepts a string of text as input, which in our case is the assembly code context surrounding a target load instruction. While source code or an intermediate representation (e.g., LLVM IR) could be used, we chose assembly code to ensure our approach is applicable to any program with a static binary executable. Static binaries can be easily disassembled into assembly code, whereas recovering source code from a binary is often challenging. We choose to set the code context for a given load instruction to consist of the 128 assembly lines preceding it and the 128 lines following it (a total of 257 lines of assembly code).

Output. The model generates a set of hints for the online prefetcher. Our design focuses on three hint types:

Prefetcher Selection Hint is an integer index specifying which sub-prefetcher in the ensemble should be used for issuing the prefetch request for a given load instruction.

Prefetch Degree Hint (Optional) is an integer that controls the aggressiveness of the selected prefetcher. The interpretation of the degree is specific to each sub-prefetcher, but a higher value always represents a more aggressive strategy. This hint is not used if the selected prefetcher does not support prefetch degree control[1].

Demand Request Filtering Hint (Optional) specifies a sub-prefetcher that should *not* receive demand requests originating from the load instruction. Filtering irrelevant demand requests reduces training noise for certain sub-prefetchers. This practice has been used in prior works [22]. Note that this hint is optional, and we do not always need to specify one for each load instruction.

4.2 PF-LLM Prompt Formatting and Training

Fine-tuning the PF-LLM model requires a large dataset of assembly code contexts paired with their corresponding ground-truth prefetching policies. We generate this dataset using the ChampSim architectural simulator [15], chosen for its wide adoption and fast simulation speed.

[1]Details on what prefetch degree is supported by each sub-prefetcher are provided in Table 2

PC	PF Type	PF Degree	AMAT
0x1234	Stride	1	25.2
0x1234	Stride	2	28.3
0x1234	Stream	1	101.0
0x1234	Stream	2	103.9
0x1235	...	...	...
...	...	...	...

Best Policy

Worst Policy

```
PF Sel: stride
PF Degree: 1
Filter: stream
```

Table 1. The dataset generation process. For each load PC, the optimal prefetcher type and degree are selected from the configuration with the best AMAT. The prefetcher to be filtered is selected from the configuration with the worst AMAT.

Our dataset generation begins with a collection of benchmark programs. For each benchmark, we run a separate ChampSim simulation for every unique combination of sub-prefetcher and prefetch degree. We modify the ChampSim simulator such that it records the Average Memory Access Time (AMAT), measured in cycles, for every load instruction identified by its unique Program Counter (PC). For example, with an ensemble of stream and stride prefetchers, each supporting degrees 1 and 2, we would run four simulations per benchmark (2 prefetchers × 2 degrees). This process yields a per-PC table of AMAT values for each prefetching policy, as illustrated in Table 1.

We use a simple deterministic heuristic to derive the ground-truth policy from this table.

1. For each load instruction with a unique PC, we select the combination of prefetcher type and degree that achieves the minimum AMAT to be the optimal prefetch policy for the load instruction.

2. For each load instruction with a unique PC, we identify the policy that results in the minimum AMAT and designate its prefetcher as the one for which demand requests should be filtered. However, to avoid disrupting certain sophisticated prefetchers with complex internal state, this filtering hint is omitted if the worst-performing prefetcher is one of the designated "advanced components" (details shown in Table 2).

This heuristic is based on two hypotheses. First, a prefetcher configuration that performs well for a given load instruction during simulation will also be effective in our final ensemble. Second, if a prefetcher performs very poorly, it means that this prefetcher is not designed for this PC's access pattern, and its internal state is likely disrupted by demand requests from that load, making it a good candidate for filtering. Further details on the dataset generation will be provided in Section 5.

Listing 1. Prompt Template for PF-LLM Training. Line 1-6 are the system prompt, Line 8-16 are the user prompt (input), Line 18-22 are the assistant prompt (output).

```
1  ──────────────────────────────────────────────
2  <|im_start|>system
3  You are a helpful assistant that generates prefetch
   hints for a given load instruction in a assembly
   code snippet.
4  The load instruction is marked with <load> and
   </load>.
5  Your output should be a JSON object with "PFSel",
   "PF Degree" and "Filter" as fields.
6  <|im_end|>
7
8  <|im_start|>user
9  movzbl 0x3adef0(%rip), %eax
10 xorl %r8d, %r8d
11 leaq 0x6dc8d(%rip), %rsi
12 <load>movq 0x10(%rsp), %rcx</load>
13 movl 0x1c(%rsp), %edx
14 movq 0x3ab11d(%rip), %rdi
15 orl $2, %eax
16 <|im_end|>
17
18 <|im_start|>assistant {
19     "PF Sel": "stride",
20     "PF Degree": 2,
21     "Filter": "stream",
22 }<|im_end|>
23 ──────────────────────────────────────────────
```

As shown in Figure 3(a), PF-LLM is trained with assembly contexts and ground-truth policies. Feeding raw assembly code to an instruction-tuned model like Qwen-2.5-Coder is suboptimal, as these models expect a specific format containing system, user, and assistant prompts. To accommodate this format, we use the prompt template shown in Listing 1, which is based on the Qwen [16] model's official template. Aligning with the instruction format has two main advantages. First, it accelerates convergence during fine-tuning. The system and user prompts (Lines 1-16) allow the base model to leverage its zero-shot capabilities to understand the task, enabling it to focus on learning the nuances of prefetching rather than the input format. Second, it facilitates structured output. Modern coding LLMs are often trained to generate structured outputs such as those in JSON format. By using this template in conjunction with structured output generation features available in serving frameworks like vLLM [20], we can enforce a strict JSON output that conforms to our specified format (Lines 18-22). This structured approach allows us to train PF-LLM effectively with our formatted data.

Noted that in our prompt template, we specify the location of the load instruction in the assembly code context through <load> and </load> tags. We have made these two tags to add special tokens in the embedding table of the PF-LLM model, such that the model can learn to recognize the

load instruction location in the assembly code context more smoothly.

4.3 PF-LLM Offline Hinting

Once trained, the PF-LLM model can generate prefetching hints for any unseen program. As shown in Figure 3(b), the process begins by disassembling a given program binary. A script then identifies all load instructions within the assembly code. For each load, its code context (128 lines before and after) is formatted using the prompt template from Listing 1 and passed to the PF-LLM model for inference.

This process requires one model inference per load instruction. However, due to the small 0.5B parameter count of PF-LLM, short context length (especially short generation length), and the high performance of the vLLM serving framework [20], the total inference cost is reasonable. A detailed cost analysis is presented in Section 5. After iterating through all load instructions, the predicted hints are collected into a single table, indexed by the load PC, and stored as a JSON file for runtime use.

4.4 LMHint Hardware Prefetcher

A key challenge is: how to transfer the offline-generated hints to be used at runtime inside the processor? To achieve this, we introduce the LMHint Prefetcher hardware prefetcher to manage sub-prefetchers at runtime with the hints provided by the PF-LLM model.

The hints generated offline are loaded into a main memory data structure called the *Prefetch Hint Table* (PHT) before program execution. The PHT is indexed by the virtual PC of each load instruction. To provide low-latency access to these hints, we introduce a small, on-chip buffer called the *Prefetch Hint Buffer* (PHB). The PHB caches the most recently used hint entries from the PHT, analogous to how a TLB caches page table entries. For our design, we use a 256-entry PHB to minimize hardware overhead while ensuring single-cycle access. We restrict the hint for each PC to a single 8-bit value, allocating 4 bits for prefetcher selection, 2 bits for prefetch degree, and 2 bits for the demand request filtering policy. Although fewer bits might suffice for an ensemble with only a small number of sub-prefetchers, we adopt an 8-bit encoding as SRAM blackboxes with row width smaller than 8-bits are not commonly available. The hint data, originally in a JSON file, is converted into this compact binary format for efficient storage in both the PHT and PHB.

When a load instruction is issued, its PC is used to look up the PHB. On a hit, the corresponding hint policy is retrieved and dispatched to control the prefetcher ensemble, as shown in Figure 3(c). The LMHint Prefetcher prefetcher is a prefetcher ensemble that orchestrates a set of conventional sub-prefetchers. The retrieved hint dictates its behavior in three ways.

1. The **prefetcher selection** hint determines which sub-prefetcher is permitted to issue prefetch requests generated in response to this load.

2. The **prefetch degree** hint is passed directly to the selected sub-prefetcher to control its aggressiveness.

3. The **demand request filtering** hint prevents the demand request from being sent to a specified sub-prefetcher, thereby protecting its internal state from irrelevant training information.

In the event of a PHB miss, a request is sent to fill the entry from the PHT in main memory. In the interim, a default prefetch policy, stored at a reserved zero-addressed entry in both the PHT and the PHB, is used.

5 Experiments

5.1 Sub-Prefetcher Selection and Configuration

We select a diverse set of well-established sub-prefetchers from the literature to form the candidates of our ensemble. These sub-prefetchers, listed in Table 2, are chosen to cover a wide spectrum of memory access patterns. Note that we exclude temporal prefetchers like Triage [35] and Triangel [2] because they are resource-intensive and designed exclusively for LLC prefetching, while our work focuses on L1D prefetching. To generate the ground-truth dataset for training PF-LLM, we implemented this entire suite of prefetchers in our ChampSim simulator. This comprehensive dataset is used for our initial exploration and to inform the selection of a smaller, cost-effective subset for our final hardware configuration. As detailed in Section 6.2, the final design includes only four of these prefetchers.

A key challenge is managing the heterogeneous prefetch degree support across these sub-prefetchers. Some components, such as Pythia [6] and Bingo [5], do not support degree control, while for others, the supported ranges vary significantly. To simplify the Prefetch Hint Table (PHT) implementation and the PF-LLM model's learning task, we normalize the prefetch degree to a tri-state variable with, in the LMHint Prefetcher framework: the prefetch degree is either 1 (conservative), 2 (moderate) or 3 (aggressive). This unified degree is then mapped to the Q1, the median, and the Q3 in the native degree range of each sub-prefetcher. For example, a LMHint Prefetcher degree of 3 (aggressive) for the DSPatch [7] prefetcher (native range 0-64) maps to $\text{round}(0.75 \times 64) = 48$. For demand request filtering, we restrict that the demand request will never be filtered out for the Pythia [6], Bingo [5], DSPatch [7], Sandbox [30], and SMS [33] prefetchers, as they are sophisticated prefetchers very brittle to demand request filtering.

5.2 ChampSim Simulation Setup

Our dataset generation and performance evaluations use a consistent system configuration, detailed in Table 3. The core configuration mimics that of an Arm Neoverse N2 core [28].

Table 2. Sub-Prefetchers and their target memory access patterns. Prefetchers marked with an asterisk (*) are excluded from demand request filtering.

Prefetcher	Target Access Pattern	PF Degree
*Pythia [6]	complex spatial access patterns	No
SMS [33]	non-contiguous, spatially access patterns within fixed-size memory regions	Yes, (range 1-31)
AMPM [17]	complex stride patterns	Yes (range 1-16)
*Bingo [5]	spatial data access patterns within page	No
Sandbox [30]	fixed offset and strided access stream patterns	Yes (range 1-8)
Power7 [29]	strided streams and streams of strides	Yes (range 0-6)
*DSPatch [7]	irregular spatial access patterns with both simple strides and non-contiguous offsets.	Yes (range 0-64)
MLOP [32]	multi-level offset patterns	Yes (range 1-16)
Stride	simple strided patterns	Yes (range 1-8)
Stream	sequential access patterns	Yes (range 1-8)
Next Line	sequential access patterns	No
PPF [8]	complex or irregular spatial patterns	No

Table 3. ChampSim Simulated System Configuration and Workload Compilation Environment.

Module	Configuration
Core	5-wide fetch, 5-wide decode 10-wide issue, 10-wide commit 120-entry IQ, 85/90-entry LQ/SQ 288-entry ROB
L1 I/D cache	64 KB each, 4-way, 64B line, 16 MSHRs PLRU, 2 cycles hit latency, **prefetch from L2**
L2 cache	512 KB, 8-way, 64B line, 32 MSHRs PLRU, 9 cycles hit latency
L3 cache	2 MB, 16-way, 64B line, 36 MSHRs PLRU, 20 cycles hit latency
DRAM	LPDDR5_5500_1x16_BG_BL32 Single channel, 1 rank per channel
Workload Compilation Environment	Ubuntu 20.04 LTS GNU GCC/G++/GFortran 9.4.0 -O2 -fpermissive -march=x86-64 200M simulation instructions per job
PF-LLM Training Parameters	8 Nvidia H20 GPUs , BF16 precision, 1e-5 learning rate, 64 effective batch size, 2 epochs

The simulation framework is built upon the DPC-3 version of the ChampSim simulator. It is worth pointing out that our workflow, shown in Figure 3, requires us to compile the SPEC benchmarks from source rather than using publicly available pre-generated traces. This is because publicly available ChampSim traces, generated using Pin [24], contain only dynamic information such as instruction execution order and memory access history but does not contain the static assembly code required by PF-LLM. By compiling the benchmarks ourselves, we gain access to the full program binaries needed for analysis. Details of our compilation environment are provided at the bottom of Table 3. These custom-compiled binaries are used for all subsequent experiments, including dataset generation, assembly context extraction, and final performance evaluation.

To ensure a rigorous evaluation, we prevent any leakage of test data into the training set. We use memory-intensive benchmarks from the SPEC 2006 suite for training the PF-LLM model and reserve the SPEC 2017 suite exclusively for testing. Thus, the PF-LLM model never sees the test set before the evaluation.

5.3 PF-LLM Training Setup

We fine-tune the PF-LLM model starting from the publicly available Qwen-2.5-Coder-Instruct [16] checkpoint. The fine-tuning process is managed using the LLaMa-factory framework [37] and the DeepSpeed [31] training system. We train the model for two epochs on 8 NVIDIA H20 GPUs. Key hyperparameters include a learning rate of 1e-5, an effective batch size of 64, and the use of the AdamW optimizer [23] with BF16 precision. The training dataset is shuffled before each epoch.

Notably, the training loss is calculated exclusively on the model's generated output (no loss is calculated on the input assembly code and the system prompt). Specifically, the loss is computed only on the tokens corresponding to the JSON output (Lines 18-22 in Listing 1). This focuses the training on the primary task of prefetch hint generation and prevents the model from being penalized for variations in reproducing the input assembly code.

6 Evaluation

This section evaluates our proposed framework. We first assess the prediction accuracy of the PF-LLM model. Next, we present the end-to-end performance of the LMHint Prefetcher prefetcher on the SPEC 2017 benchmark suite. We then analyze the source of the performance gains and conclude with a comprehensive analysis of the offline and runtime overheads.

6.1 PF-LLM Model Accuracy

We evaluate the PF-LLM model's ability to predict the optimal prefetching policy from static assembly code. The model

is trained on data generated from the SPEC 2006 benchmark suite. Figure 4 shows the training loss and validation accuracy curves. The loss decreases consistently while the accuracy improves, with the model reaching a final prediction accuracy of 95.0% on the held-out test set. We ceased training at this point due to compute and time constraints, but we believe that with a larger training dataset, the accuracy could improve further, leading to an even closer approximation of the optimal prefetching policy. Nonetheless, a 95% accuracy is a strong indicator of the LLM's capability in this domain and, as we will show, is sufficient to achieve significant performance improvements.

Figure 5 provides a more detailed analysis of the model's predictions. The ground-truth data in Figure 5(b) reveals that the optimal prefetcher policy is not uniformly distributed. Certain policies, such as the Sandbox [30] prefetcher with a moderate degree, are optimal far more frequently than others. The confusion matrix in Figure 5(a) demonstrates that the PF-LLM model successfully learns and reproduces this skewed distribution. The strong diagonal across the matrix indicates a high rate of correct predictions. Crucially, our analysis of the mispredictions (the off-diagonal elements) reveals that when PF-LLM mis-predicts, its output is not random but highly likely to be the second-best option, yielding performance close to the optimum. This finding suggests that the model learns not just the single best policy but also the general characteristics of effective prefetching for a given code context, further confirming its ability to provide high-quality hints to the hardware prefetcher.

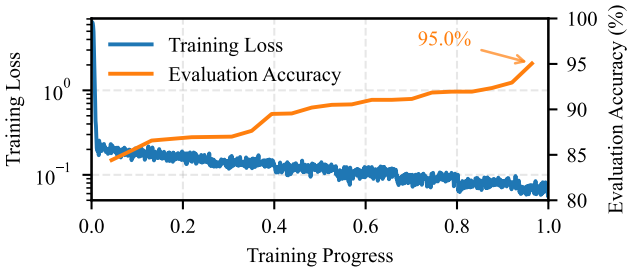

Figure 4. PF-LLM model training convergence. The final evaluation accuracy is 95.0%, showcasing the effectiveness of using a large language model to hint hardware prefetchers.

6.2 Performance of the LMHint Prefetcher

We now evaluate the end-to-end performance of the LMHint Prefetcher prefetcher, guided by hints from the PF-LLM model, on the memory-intensive SPEC 2017 benchmarks. We conduct an ablation study to quantify the contribution of each hint type. We evaluate the following four configurations.

LMHint-S: The baseline LMHint configuration, using only the prefetcher **S**election hint.

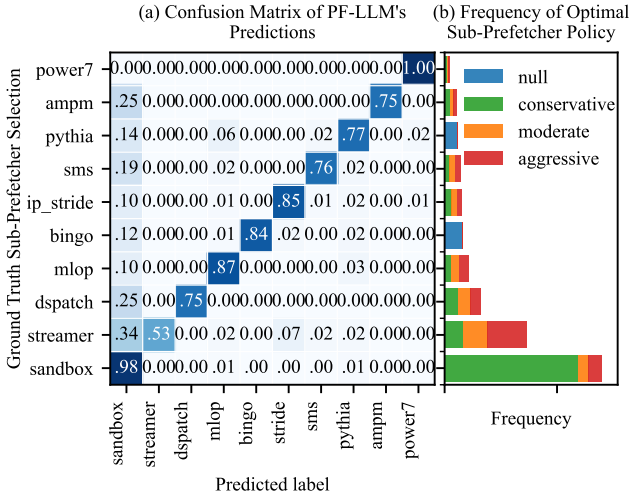

Figure 5. PF-LLM's performance evaluation. (a) Confusion matrix of PF-LLM's predictions. (b) Frequency of optimal sub-prefetcher policy.

LMHint-SD: Adds prefetch **D**egree control to the selection hint.

LMHint-SDF: Adds demand request **F**iltering to the selection and degree hints.

LMHint-SDFR: A **R**educed-cost configuration that uses all three hint types but restricts hardware of the prefetcher ensemble to include only the four most frequently chosen sub-prefetchers identified in Figure 5(b).

Figure 6 presents the detailed IPC speedup for each benchmark, and Figure 7 summarizes the geometric mean speedups over a no-prefetch baseline. We draw the following conclusions from these results.

First, all LMHint Prefetcher configurations significantly outperform existing state-of-the-art single prefetchers and ensemble methods across most benchmarks. Our full-featured configuration, **LMHint-SDF**, achieves a geometric mean IPC improvement of **9.8%** over the best-performing single prefetcher (Sandbox) and **18.9%** over the best-performing prior ensemble method (Alecto). This represents a substantial, cross-generation, performance gain.

Second, the ablation study demonstrates the value of each hint type. The progression from LMHint-S to LMHint-SDF shows incremental performance improvements. Adding prefetch degree control (LMHint-SD) provides a 0.3% average IPC gain over selection alone, and subsequently adding demand request filtering (LMHint-SDF) contributes another 0.3% gain. Although prefetcher selection provides the largest benefit, these fine-grained controls are crucial for maximizing performance.

Third, the reduced-cost LMHint-SDFR configuration performs remarkably well, slightly outperforming the LMHint-SDF configuration by 0.01% on average. While this small

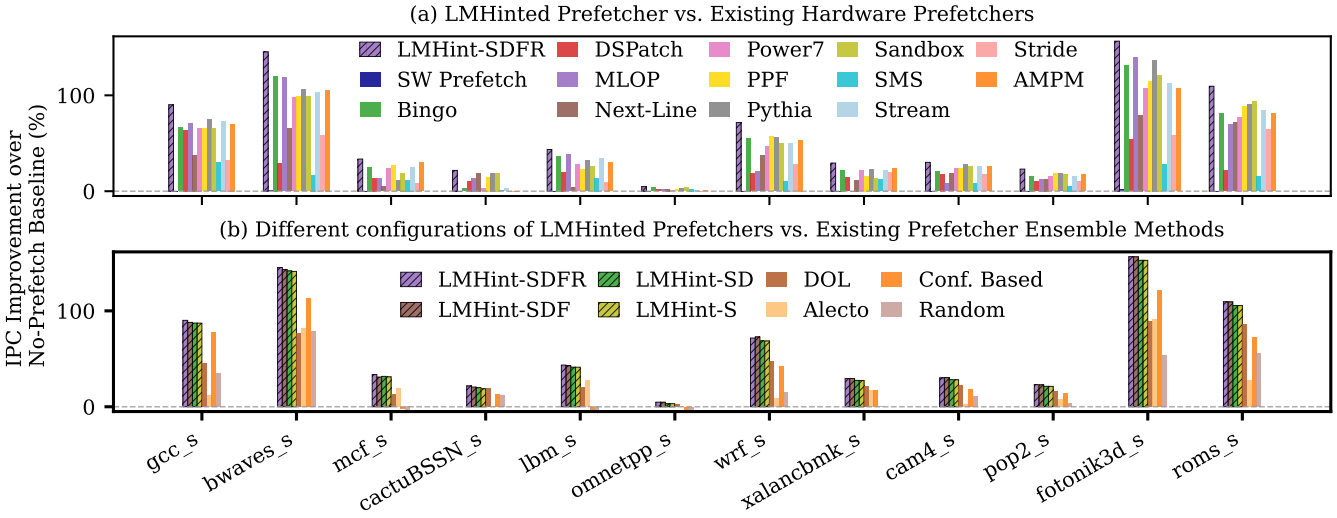

Figure 6. Performance Evaluation of the LMHint Prefetcher. (a) compares the performance of the LMHint Prefetcher in terms of IPC over No Prefetch with existing state-of-the-art hardware prefetchers. (b) compares the performance against existing state-of-the-art hardware prefetcher ensemble methods. **SW Prefetch** stands for software prefetching [1]; **Conf. Based** stands for confidence-based prefetcher ensemble; **Random** stands for the naive baseline of random prefetcher selection.

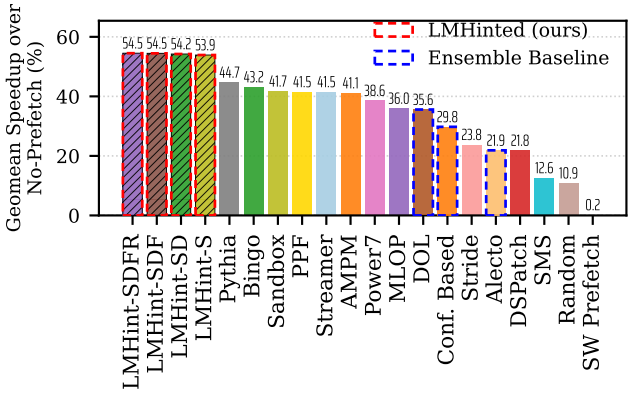

Figure 7. Geometric Mean Speedup vs. No Prefetch (%) for different prefetcher configurations for **SPEC 2017** benchmarks. This figure shares color legends with Figure 6.

gain is within the noise margin, it is a significant result as it demonstrates that we can reduce the hardware complexity and area by implementing only four sub-prefetchers instead of eleven, without sacrificing performance. This reduction is achieved without retraining the PF-LLM model. We simply constrain the JSON schema used by the vLLM inference engine at hint generation time to only allow the model to select from the four chosen prefetchers.

Finally, our approach successfully addresses a known challenge in prefetcher ensemble design. As shown in Figure 7, conventional online ensemble methods like Alecto [22] and DOL [19] perform worse than the best single prefetchers. This performance degradation is a known limitation when such PC-centric online policies are used to manage advanced, non-PC-localized components like spatial prefetchers. Because these sub-prefetchers build intricate state by tracking patterns across entire memory regions, not single instruction streams, the PC-based selection logic of methods like Alecto disrupts their pattern detection by either polluting their training with irrelevant demand requests or filtering out key demand requests. LMHint Prefetcher avoids this problem by using static, context-aware hints. The ability to filter demand requests is particularly crucial, as it shields the delicate state of these advanced prefetchers from such disruptive training inputs. This effective orchestration allows the ensemble to function cohesively, leading to a performance superior to all baselines.

6.3 Evaluation on Real-World Web Serving Workloads

To validate the practicality of our approach beyond synthetic benchmarks, we evaluate LMHint Prefetcher on real-world web serving workloads from three of the mostly widely-used open-source web applications: Apache HTTP Server [12], MySQL Database [26], RocksDB Key-Value Store [10] and Xapian Search Engine [34]. We trace and simulate these applications under realistic workloads using the same methodology as our prior benchmark with one exception such that we increase the main memory latency by 2× to better reflect the crowded micro-service co-serving environments in modern data centers [14]. The evaluation results in Figure 8 show that LMHint Prefetcher also remains effective in these real-world scenarios and also performs better than all existing single prefetcher and ensemble baselines.

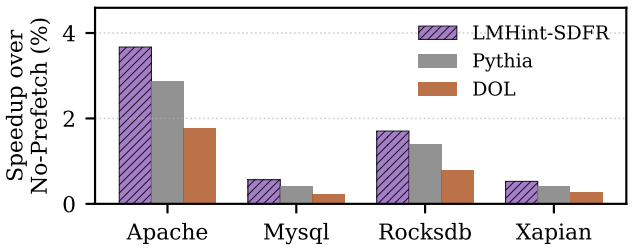

Figure 8. Speedup vs. No Prefetch (%) for different prefetcher configurations for **Real-World Web Serving** workloads. This figure shares color legends with Figure 6.

It is worth noting however, that the performance gains of these web serving workloads are more modest compared to the SPEC benchmarks. We attribute this to two factors. First, these applications are generally more I/O-bound rather than CPU/memory-bound, limiting the potential impact of prefetching optimizations. Second, these applications have already undergone extensive manual optimizations over many years, many even implement their own caching and prefetching mechanisms, leaving less room for improvement through auto-prefetching.

6.4 Analysis of Performance Improvement

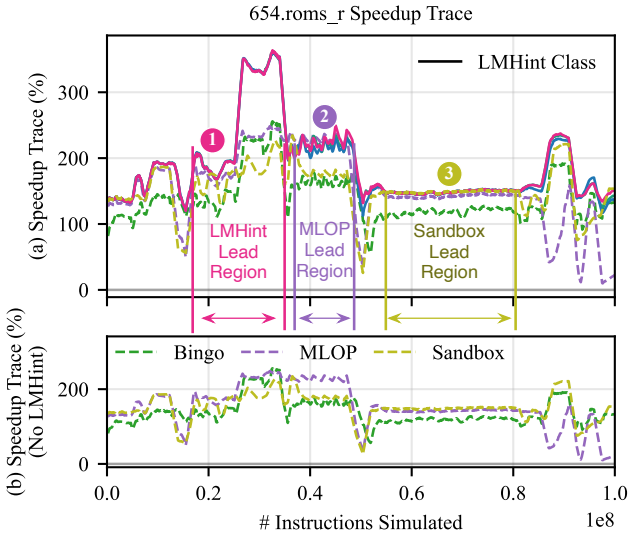

Figure 9. Speedup Trace (in terms of IPC improvement over no-prefetch baseline) for workload 654.roms. This figure shares color legends with Figure 6. (b) shows the heartbeat IPC trace for baselines, (a) adds the LMHint Prefetcher traces on top of it. Three regions ①, ②, and ③ are highlighted to show how LMHint Prefetcher outperforms the baselines.

In Figure 9, it is the **heartbeat IPC trace** of 654.roms_r that visually proves PF-LLM delivers **oracle-level adaptation** with **zero runtime adaptation latency**.

In one glance at Figure 9, it shows LMHint Prefetcher **tracking the upper envelope** of every sub-prefetcher, while others oscillate and lag behind. This is because monolithic prefetchers are inherently limited because they are committed to a single, fixed policy, making them impossible to adapt to the program's changing memory access patterns. In contrast, LMHint Prefetcher is proactive. By leveraging per-PC hints generated offline by PF-LLM, the LMHint Prefetcher knows the most effective policy for a code region before it is executed. As the program's control flow moves between phases, LMHint Prefetcher instantaneously switches policies without any runtime trial-and-error. This allows its IPC trace in Figure 9(a) to closely track the upper performance envelope of all its constituent sub-prefetchers. This oracle-like selection is evident in regions ② and ③, where LMHint Prefetcher immediately matches the performance of the best-performing prefetcher for that phase, MLOP and Sandbox respectively.

Furthermore, LMHint Prefetcher can achieve performance superior to any single sub-prefetcher. In region ①, the program's memory access pattern is too diverse for any individual prefetcher to handle effectively. Here, the advantage of LMHint Prefetcher's fine-grained control becomes apparent. LMHint Prefetcher uses its selection, degree, and demand request filtering hints to orchestrate the sub-prefetchers at the individual load-instruction level. This allows each sub-prefetcher to focus only on the access streams it is designed for, minimizing interference from unrelated demand requests. Consequently, the ensemble's performance becomes additive, exceeding the maximum performance of any single component. This proactive, per-PC orchestration effectively realizes the benefits of an oracle that possesses perfect, fine-grained knowledge of the optimal policy for each phase of program execution.

6.5 Overhead Analysis

Our approach introduces two primary overheads: the one-time offline inference cost to generate hints using the PF-LLM model and the hardware cost for the storage of the hints.

Offline Inference Cost. Generating hints requires running one PF-LLM inference for each load instruction in a program binary. While this may sound expensive, our experiments show the cost is practical. We use the vLLM [20] framework to serve the PF-LLM model. Figure 10(a) shows that on 1 NVIDIA H20 GPU, our system achieves an inference throughput of up to 234 requests per second. This high performance is due to three factors: 1) the small 0.5B parameter count of PF-LLM; 2) the input context lengths are relatively short, with most assembly contexts being under 3000 tokens, as shown in Figure 10(b). 3) our methodology of analyzing all load instructions in a binary provides a large number of independent requests. This allows us to fully leverage vLLM's batching capabilities to maximize hardware utilization. As Figure 10(a) illustrates, throughput increases significantly

with higher concurrency. Our workload can easily supply a batch size sufficient to saturate the GPU's computational capacity, achieving a substantially higher throughput than processing requests serially.

This high throughput translates to a manageable total hinting time. As shown in Figure 10(c), the total inference time scales linearly with the binary size. This linear relationship is expected because the total number of inferences required is determined by the number of load instructions, which is directly proportional to the size of the code. For the entire SPEC 2017 suite, generating hints for all benchmarks requires 38.5 minutes on our 8-GPU system. This one-time cost is comparable to the 25.4 minutes required to compile the suite on a 16-core machine, placing it well within acceptable limits for a typical software development and deployment pipeline.

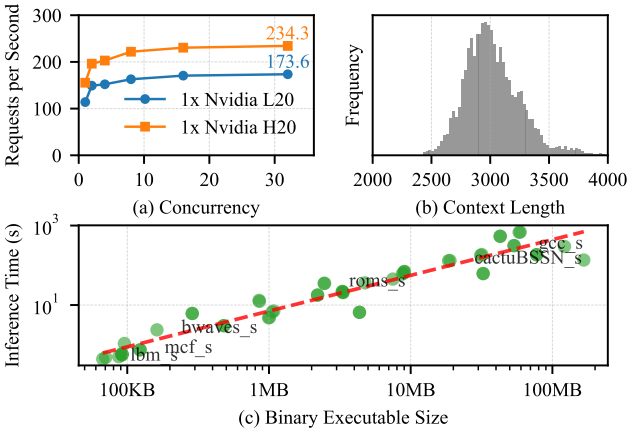

Figure 10. PF-LLM inference overhead analysis. (a) Inference throughput (requests per second) on different GPU types. (b) Distribution of input assembly context lengths. (c) Total inference time versus application binary size for SPEC 2017 benchmarks.

Runtime Storage Cost. The generated hints are stored in main memory in a Prefetch Hint Table (PHT). Each entry in the PHT stores a 48-bit virtual PC (given x86-64 architecture) and our 8-bit hint, totaling 56 bits (7 bytes) per load instruction. We analyzed the SPEC 2017 binaries and found an average of 10.62K load instructions per megabyte of executable code. This translates to a PHT storage overhead of 74.34 KB per MB of the binary, or a 7.26% increase in the static program footprint.

We argue that the offline inference and storage overheads of our approach are a justifiable trade-off for the substantial 10-20% IPC improvement it delivers. Achieving a similar performance uplift through conventional hardware enhancements, such as widening the core, deepening the pipeline, or enlarging caches, would incur significantly higher area, power, and design complexity overheads.

7 Discussion

In this section, we delve into the rationale behind the key design choices of PF-LLM and analyze its position within the broader landscape of prefetching techniques. We begin by justifying our strategic choice of assembly code as the input medium, highlighting its universality and close alignment with runtime behavior compared to source code. Next, we contrast PF-LLM with Profile-Guided Optimization (PGO) and traditional software prefetching, emphasizing how our hybrid approach achieves robustness against input variations while avoiding the overhead of instruction injection. We then assess the generalizability of our framework across different Instruction Set Architectures (ISAs) and machine configurations. Finally, we address current limitations regarding JIT compilation and ASLR, and conclude by exploring the broader implications of LLM-guided microarchitecture for future hardware-software co-design.

7.1 Assembly Code vs. Source Code as Input:

Our PF-LLM takes input from the assembly code directly stripped from the binary executable. We chose assembly-level input for PF-LLM due to its universality and ability to manage context complexity. First, source code or intermediate representations (IR) are not always available, particularly for closed-source binaries. Second, although stripping code directly from the binary executable may seem to break inter-procedural calling relationships, compilers often place functions with high call affinity into adjacent memory regions to improve instruction cache performance. This allows our directly extracted context without regard for function boundaries to capture inter-functional information. Conversely, function inlining or loop unrolling would cause the context length to grow exponentially, making it difficult for our LLM to capture meaningful semantic patterns within a fixed-context-length. Finally, we believe that assembly code aligns most closely with the runtime context. Additional layers of indirection make it harder for LLMs to learn the mappings between the input abstraction and the output hints that are directly related to the runtime context. Nevertheless, our experiments confirm that assembly-code is sufficiently expressive for the LLM to generate effective hints.

7.2 Software Prefetch vs. Profile-Guided Optimization vs. Hardware Prefetching:

A key advantage of PF-LLM over Profile-Guided Optimization (PGO) is its robustness to input data variations. PGO relies on profiling runs with specific inputs, which may not represent the deployment workload, leading to overfitting. Although the hints from the offline PF-LLM are static for a specific binary, our overall approach is adaptive to the runtime environment. Consistent with prior work [4], we find that memory-access patterns are primarily determined by the program code, meaning the pattern for a given load

instruction largely reflects the code's semantics across runs. With our sufficiently large context window, the PF-LLM can capture and distinguish different input- or branch-dependent cases. The selected prefetcher can continue to learn from runtime data to handle remaining data- and input-dependent variation. A key contribution of our work is the definition of a clear and generalizable interface between the offline hint generation and the online prefetcher selection. This synergy enables static, optimized, semantic-aware hints to guide dynamic, runtime-adaptive hardware.

Furthermore, as shown in Figure 6 and Figure 7, the software prefetch [1] exhibits poor performance compared to hardware prefetching techniques[2]. This highlights the limitations of purely software-based approaches that inject prefetch instructions directly into the instruction stream. Such approaches not only have limited visibility into runtime memory access patterns but also increase the instruction counts leading to additional overhead at the core's front-end.

7.3 Generalizability and Robustness:

Instruction Set Architecture (ISA) Dependency: While our evaluation focuses on x86-64, the methodology of PF-LLM is ISA-agnostic. For different ISAs, our model could be retrained or fine-tuned to adapt to the specific target ISA.

Machine Configuration Dependency: Optimal prefetching policies can vary with micro-architectural parameters like cache size or bandwidth. Although our current model is trained for a specific configuration, the framework is extensible. To support diverse hardware, the model could be retrained with data from different configurations, or hardware parameters could be included as additional prompts to the LLM, enabling it to generate configuration-aware hints. We envision that CPU vendors would supply the PF-LLM model, as they possess detailed knowledge and accurate simulators to generate data to train the PF-LLM model for their specific platforms. Future work may also focus on training a single, generalizable model capable of adapting to diverse machine configurations for prefetch hinting.

7.4 Limitations:

We acknowledge two primary limitations of the current PF-LLM implementation. First, it does not natively support programs requiring Just-In-Time (JIT) compilation or programs that execute on top of some intermediate bytecode (e.g., Java), as PF-LLM relies on analyzing static binaries. However, this can be addressed by extending our approach to analyze the static Intermediate Representation (IR) which is available before runtime, allowing us to train a separate LLM model to generate hints from the IR. Second, our current prototype does not account for Address Space Layout Randomization (ASLR). Since PF-LLM generates hints indexed by static Program Counters (PCs), runtime randomization could break the mapping between hints and instructions. This can be resolved by integrating the hinting mechanism with the OS loader. By applying the same randomization offset to the hint table as is applied to the code segment, the correspondence between hints and runtime PCs can be preserved.

[2]Software prefetching was evaluated by collecting a standalone software prefetch instruction trace and then feeding it back as prefetch requests into the ChampSim simulator to measure performance.

7.5 Implications:

The success of PF-LLM suggests a broader paradigm of LLM-guided microarchitecture. We believe the fundamental approach of using an LLM to comprehend static code context offline can provide valuable guidance to other dynamic hardware mechanisms. Future work could explore applying this co-design methodology to other classic challenges. For instance, an LLM could analyze complex control-flow structures to provide hints for branch predictors or infer data reuse characteristics to inform more intelligent cache replacement and insertion policies. This direction promises a new class of hybrid hardware-software co-optimizations that leverage the deep analytical power of LLMs to enhance the efficiency of runtime hardware.

8 Conclusion

We introduced PF-LLM, a fine-tuned Large Language Model that analyzes static assembly code to generate prefetching hints for a lightweight ensemble hardware prefetcher named LMHint Prefetcher. By offloading complex policy decisions to offline analysis, our approach eliminates costly online training. Our evaluation demonstrates that PF-LLM predicts the optimal prefetching policy with 95% accuracy. This translates to a significant performance gain, with an average IPC improvement of 9.8% over a state-of-the-art single prefetcher and 18.9% over leading ensemble methods on memory-intensive SPEC 2017 benchmarks.

This work pioneers a co-design methodology where offline AI models direct hardware during runtime. The success of this LLM-guided approach suggests the potential of using AI models to guide other dynamic microarchitectural mechanisms in advancing processor performance.

Acknowledgments

This research was partially conducted by the ACCESS AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government. It was also partially supported by the Research Grants Council of Hong Kong SAR (16213824 & 16212825) and the Research Grants Council YCRG Grant of Hong Kong SAR (C6003-24Y).

References

[1] Sam Ainsworth and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 305–317. doi:10.1109/CGO.2017.7863749

[2] Sam Ainsworth and Lev Mukhanov. 2024. Triangel: A High-Performance, Accurate, Timely On-Chip Temporal Prefetcher. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Los Alamitos, CA, USA, 1202–1216. doi:10.1109/isca59077.2024.00090

[3] Alaa R. Alameldeen and David A. Wood. 2007. Interactions Between Compression and Prefetching in Chip Multiprocessors. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, USA, 228–239. doi:10.1109/HPCA.2007.346200

[4] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526. doi:10.1145/3373376.3378498

[5] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 399–411. doi:10.1109/HPCA.2019.00053

[6] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, USA, 1121–1137. doi:10.1145/3466752.3480114

[7] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. DSPatch: Dual Spatial Pattern Prefetcher. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 531–544. doi:10.1145/3352460.3358325

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

[9] George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. 2010. Resource-Aware Compiler Prefetching for Many-Cores. In *2010 Ninth International Symposium on Parallel and Distributed Computing*. IEEE, Los Alamitos, CA, USA, 133–140. doi:10.1109/ISPDC.2010.16

[10] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[11] Babak Falsafi and Thomas F. Wenisch. 2014. *A Primer on Hardware Prefetching*. Morgan & Claypool Publishers, San Rafael, CA, USA.

[12] Roy T. Fielding and Gail Kaiser. 2002. The Apache HTTP server project. *IEEE Internet Computing* 1, 4 (2002), 88–90.

[13] Gerasimos Gerogiannis and Josep Torrellas. 2023. Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 698–713. doi:10.1145/3613424.3623780

[14] Saverio Giallorenzo, Jacopo Mauro, Martin Gyde Poulsen, and Filip Siroky. 2021. Virtualization costs: benchmarking containers and virtual machines against bare-metal. *SN Computer Science* 2, 5 (2021), 404.

[15] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR] https://arxiv.org/abs/2210.14324

[16] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186

[17] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 499–500. doi:10.1145/1542275.1542349

[18] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 247–259.

[19] Sushant Kondguli and Michael Huang. 2018. Division of Labor: A More Effective Approach to Prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 83–95. doi:10.1109/ISCA.2018.00018

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA.

[21] Mengming Li, Qijun Zhang, Yichuan Gao, Wenji Fang, Yao Lu, Yongqing Ren, and Zhiyao Xie. 2025. Profile-Guided Temporal Prefetching. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 572–585.

[22] Mengming Li, Qijun Zhang, Yongqing Ren, and Zhiyao Xie. 2025. Integrating Prefetcher Selection with Dynamic Request Allocation Improves Prefetching Efficiency . In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 204–216. doi:10.1109/HPCA61900.2025.00026

[23] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, New Orleans, LA, USA. https://openreview.net/forum?id=Bkg6RiCqY7

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. doi:10.1145/1065010.1065034

[25] Sparsh Mittal. 2016. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv.* 49, 2, Article 35 (Aug. 2016), 35 pages. doi:10.1145/2907071

[26] Oracle Corporation 2025. *MySQL 8.4 Reference Manual*. Oracle Corporation. https://dev.mysql.com/doc/refman/8.4/en/ Accessed: 2025-12-19.

[27] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 118–131. doi:10.1109/ISCA45697.2020.00021

[28] Andrea Pellegrini. 2021. Arm Neoverse N2: Arm's 2 nd generation high performance infrastructure CPUs and system IPs. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, IEEE, Los Alamitos, CA, USA, 1–27.

[29] David Prat, Cristobal Ortega, Marc Casas, Miquel Moretó, and Mateo Valero. 2015. Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7. arXiv:1501.02282 [cs.DC] https://arxiv.org/abs/1501.02282

[30] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 626–637. doi:10.1109/HPCA.2014.6835971

[31] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703

[32] Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Multi-Lookahead Offset Prefetching. In *The Third Data Prefetching Championship (DPC-3)*. IEEE, Los Alamitos, CA, USA.

[33] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, USA, 252–263. doi:10.1109/ISCA.2006.38

[34] Xapian Development Team. 2025. Xapian: Open Source Search Engine Library. https://xapian.org/ Accessed: 2025-12-19.

[35] Hao Wu, Krishnendra Nathella, Matthew Pabst, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2022. Practical Temporal Prefetching With Compressed On-Chip Metadata. *IEEE Trans. Comput.* 71, 11 (2022), 2858–2871. doi:10.1109/TC.2021.3065909

[36] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. 2024. RPG2: Robust Profile-Guided Runtime Prefetch Generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 999–1013. doi:10.1145/3620665.3640396

[37] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Association for Computational Linguistics, Bangkok, Thailand. http://arxiv.org/abs/2403.13372