GraFetch: Accelerating Graph Applications Through Domain Specific Hierarchical Hybrid Prefetching

Pengmiao Zhang, Rajgopal Kannan, and Viktor K. Prasanna, *Fellow, IEEE*

***Abstract*—Memory performance bottlenecks the execution of graph applications, from traditional graph analytics (GA) to rapidly evolving graph neural networks (GNNs), due to the large size and complexity of graphs. While machine learning (ML) algorithms have shown potential in data prefetching to hide memory access latency, existing approaches face challenges with phase transitions and irregular memory access patterns in graph applications. To address these challenges, we introduce GraFetch, a specialized prefetching system for accelerating graph applications. GraFetch comprises of 1) a novel Hierarchical Hybrid Prefetching (HHP) framework that supports the cooperation of phase-specific ML predictors for high-complexity pattern prefetching and rule-based prefetchers for low-complexity pattern prefetching; and 2) Domain Specific Machine Learning (DSML) models integrated in the framework, which incorporate domain knowledge of graph applications to detect phases, recognize patterns, and predict memory accesses. We evaluate our approach using popular GA frameworks GPOP and X-Stream, and state-of-the-art GNN frameworks PyG and DGL. Our domain specific attention-based memory access predictors achieve 7.4% higher F1-score for delta (consecutive address jump) prediction and 15.35% higher accuracy@10 for page prediction compared with basic attention models. GraFetch achieves an average IPC improvement of 12.47% for GA and 4.18% for GNNs over a system with no prefetcher. This outperforms state-of-the-art rule-based prefetchers BO (7.12% for GA, 1.10% for GNNs), ISB (3.82% for GA, 1.60% for GNNs), and IMP (8.47% for GA, 2.20% for GNNs), as well as ML-based prefetchers Voyager (9.61% for GA, 3.14% for GNNs) and TransFetch (10.98% for GA, 2.48% for GNNs).**

***Index Terms*—Machine learning, prefetcher, domain specific, graph analytics, graph neural networks.**

I. Introduction

GRAPHS are widely used across various fields, ranging from social networks and biological systems to recommendation engines, owing to their capacity to represent complex relationships and interactions among entities intricately. Graph applications use graph structures and algorithms to solve real-world problems. Traditional Graph Analytics (GA) [1] focus on extracting patterns directly from the graph structure using algorithms such as Breadth-First Search, Connected Components, and PageRank. Researchers have proposed various effective and scalable graph processing frameworks to improve graph analytics performance, such as GPOP [2] and X-Stream [3]. Recently, Graph Neural Networks (GNNs) [4] have risen as a crucial breakthrough in graph algorithms, offering advanced techniques for learning representations from graph-structured data. Their ability to encode graph topology and node features into low-dimensional embeddings has revolutionized tasks such as node classification and link prediction [5]. Frameworks for GNN, such as PyTorch Geometric (PyG) [6] and Deep Graph Library (DGL) [7], provide the necessary infrastructure and tools for implementing, training, and evaluating GNN models efficiently.

Graph applications can be efficiently developed based on the GA and GNN frameworks. However, they often encounter performance bottlenecks due to low memory performance [8]. Studies show that up to 60% of execution time in GNN workloads is spent on memory-bound operations, such as feature aggregation and neighborhood sampling [9]. Frequent, irregular memory accesses to large graph data, compounded by parallel execution randomness, create a bottleneck that limits computational efficiency and scalability [10]. These insights motivate us to focus on memory optimization as a critical step toward improving the performance of graph applications.

Data prefetching [11] is a memory optimization technique that hides memory access latency. It predicts and fetches data from main memory into the cache hierarchy before processor requests. There are several prefetching mechanisms for accelerating graph applications. Traditional rule-based prefetchers, for example, rely on the heuristic spatial locality [12], [13], [14] and temporal locality [15], [16] of memory accesses. These prefetchers are effective for regular stride and streaming patterns [17], and their simplicity allows for straightforward hardware implementation. However, these rule-based prefetchers cannot handle the complex memory access patterns seen in graph applications. On the other hand, Machine Learning (ML) algorithms have shown high performance in recent memory access prediction and prefetching tasks [18], [19], [20], [21], [22]. In particular, models using the attention mechanism [23] have demonstrated high accuracy in complex memory access pattern detection and high practicality due to their parallelizable structure [8], [24].

Considering the ubiquity and significance of graph analytics and graph neural networks, we believe there is a need to develop dedicated high-performance ML prefetchers for such

Received 26 June 2024; revised 12 May 2025; accepted 14 May 2025. Date of publication 29 May 2025; date of current version 20 June 2025. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CNS-2009057 and Grant SaTC-2104264, and in part by the DEVCOM Army Research Lab (ARL) under Grant W911NF2420194. Recommended for acceptance by M. Gokhale. *(Corresponding author: Pengmiao Zhang.)*

Pengmiao Zhang and Viktor K. Prasanna are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90007 USA (e-mail: pengmiao@usc.edu; prasanna@usc.edu).

Rajgopal Kannan is with DEVCOM ARL Army Research Office, Los Angeles, CA 90089 USA (e-mail: rajgopal.kannan.civ@army.mil).

Digital Object Identifier 10.1109/TPDS.2025.3575106

applications. However, two major shortcomings in existing ML-based prefetchers hinder the advancement of such specialized solutions. *1) The fallacy of one-model-fits-all.* Graph applications typically involve multiple distinct phases. For instance, the GA framework X-Stream utilizes Scatter and Gather phases in each processing iteration, while the GNN framework PyG employs Aggregate and Update phases in each forwarding computation step. The memory access patterns vary significantly across these phases, making it ineffective to train a single general ML-based predictor that performs well for all phases [25]. Moreover, within each phase, regular and irregular memory patterns are interleaved. For instance, loading a feature vector for one graph node constitutes regular streaming access, whereas visiting neighboring nodes may result in irregular memory address jumps. The ease of learning regular patterns often dominates model training [26], resulting in a biased model that excels at predicting regular patterns but performs poorly on complex ones, contradicting our motivation behind applying ML to prefetching. *2) Limited incorporation of domain knowledge.* Existing ML-based prefetchers [19], [22], [24], [27] model the prefetching process as a simple sequence prediction problem, lacking integration of domain knowledge from architecture or computation. However, the domain knowledge is crucial for modeling. The context of the architecture shapes the design of ML prefetching models. For example, in multi-core systems, program counters (PCs) hold greater significance compared to single-core systems, as they specify the core for a memory request. The context of the computation influences the design and training of an ML-based prefetcher. For example, the paradigm of a graph application defines processing phases, enabling training multiple phase-specific models; traversing nodes across pages causes wide-ranging page jumps, motivating spatial-temporal prefetching; the information on graph inputs and GNN models can offer valuable insights and boost model performance.

To address the above shortcomings, we propose GraFetch, accelerating Graph applications via preFetching. Specifically, GraFetch consists of 1) a novel *Hierarchical Hybrid Prefetching (HHP) framework*, which analyzes memory access patterns and allocates different prefetchers accordingly, and 2) *Domain Specific Machine Learning (DSML) models for prefetching* with domain knowledge integrated into the ML-based components in HHP.

The Hierarchical Hybrid Prefetching (HHP) framework targets graph applications with multiple processing phases and complex memory access patterns. We design four levels of memory access analysis in HHP, consisting of: 1) application level, where memory accesses are extracted; 2) phase level, where phases are identified based on the computation paradigm; 3) snippet level, where short sequences sharing similar patterns are detected and specific prefetchers are allocated based on snippet pattern complexity; and 4) access level, where future memory accesses are predicted and prefetches are issued. HHP offers several advantages over existing approaches. First, HHP employs phase-specific ML models, which outperform a one-model-fits-all model approach. Second, HHP can better exploit the capabilities of ML models by training ML models using snippets under complex patterns. Third, HHP offers flexibility in integrating lightweight rule-based prefetchers to handle regular patterns, thereby saving computational resources. Moreover, analyzing snippet complexity can pinpoint the pain points of the memory performance of an application, guiding the software-level optimizations and prefetching strategies. This framework distinguishes this work from our previous work [8], which uses purely ML-based predictors for prefetching.

To develop components within the HHP framework, we introduce a *DSML for prefetching* methodology. Following the methodology, we optimize the DSML-based memory access predictors by incorporating domain knowledge of graph applications, including: *modality*, *structure*, *phase*, and *locality*. 1) Modality. Modality refers to the way of describing an event [28]. In a multi-core system, we treat the program counter (PC) sequence as a distinct modality input in a prediction model, introducing the perspective of the requested core to a memory access. 2) Structure. We incorporate the domain features from the graph structures and the metadata of the GNN model to further improve the model performance. By merging memory access addresses, program counters, and structural features, we propose a novel model backbone DAMMA, a Domain Specific Attention-Based Multi-Modality Network Using Attention Fusion. 3) Phase. Due to the distinct memory access patterns across phases in graph applications, we propose training phase-specific predictors. To support this, we design phase transition detectors using a novel *soft transition mechanism* that avoids false positives. 4) Locality. Wide-range memory page jumps in graph applications motivates us to develop page and delta predictors for temporal and spatial locality, respectively. We implement a *chain spatio-temporal prefetching* strategy to manage ML model predictions. In addition, we propose a composite entropy to evaluate snippet complexity and train DAMMA-based predictors on complex snippets exclusively. At run time, low-complexity snippets are handled by rule-based prefetchers, while high-complexity snippets are allocated to ML-based predictors.

Our prefetcher relies on several essential hardware capabilities such as fast inferencing and efficient online training. There is potential for deployment in future systems enabled by emerging hardware technologies. For example, embedded FPGAs (eFPGAs) or other lightweight inference accelerators that are tightly coupled with memory controllers are promising for supporting such requirements [29]. These technologies facilitate rapid access to model parameters stored in on-chip memory, enable low-latency inference through highly parallel hardware execution, and support efficient model updates via high-bandwidth communication.

Our main contributions are summarized as follows:

- We propose a novel Hierarchical Hybrid Prefetching (HHP) framework that analyzes memory access patterns through four levels—application, phase, snippet, and access—and allocates different prefetchers.
- We introduce a methodology for developing Domain Specific Machine Learning (DSML) models for prefetching, which captures the context of the architecture and the computation of an application.
- We propose GraFetch to exemplify our approach in the domain of graph applications. We propose optimizations

including: phase-specific page predictors and delta predictors using a novel DAMMA structure trained on high-complexity snippets, a phase detector with a soft transition mechanism, an unsupervised snippet detector (SoftKSWIN), and a composite entropy for measuring memory sequence complexity and prefetcher allocation.
- We evaluate our approach using popular frameworks of graph analytics (GA) GPOP and X-Stream, and state-of-the-art graph neural networks (GNNs) frameworks PyG and DGL. Results show that our optimized DAMMA-based models achieve 7.4% higher F1-Score for delta prediction and 15.35% higher accuracy@10 for page prediction compared with the basic models on average.
- GraFetch achieves an average IPC improvement of 12.47% for GA and 4.18% for GNNs over a system with no prefetcher. This outperforms rule-based prefetchers BO (7.12% for GA, 1.10% for GNNs), ISB (3.82% for GA, 1.60% for GNNs), and and IMP (8.47% for GA, 2.20% for GNNs), as well as ML-based prefetchers Voyager (9.61% for GA, 3.14% for GNNs) and TransFetch (10.98% for GA, 2.48% for GNNs).

II. Related Work

A. Machine Learning for Data Prefetching

Machine Learning (ML) has recently achieved success in memory access prediction and data prefetching. One indirect approach involves enhancing existing prefetchers, aiding in decisions like prefetcher configuration and filtering [30]. Liao et al. [31] utilize nearest neighbor, support vector machine, and decision tree for optimizing data center prefetching configurations. Rahman et al. [32] optimize a subset of prefetchers to maximize prefetcher effectiveness. Zhang et al. introduce RAOP [33], employing a recurrent neural network to improve an existing offset prefetcher. Recent works [20], [34] explore using reinforcement learning to configure prefetchers. A direct way of applying ML for prefetching involves framing memory access prediction as a sequence prediction problem. Peled et al. propose NN-pref [35] uses a multi-layer perceptron to detect arbitrary memory access patterns. Long Short-Term Memory (LSTM) network shows high accuracy in predicting memory accesses [22], [27], [36], [37]. Recently, the attention mechanism [23] have demonstrated significant improvements in ML-based prefetching. Voyager [19] combines LSTM and the attention mechanism to enhance predictive capabilities. TransFetch [24] achieves state-of-the-art prefetching performance using a fully attention-based network. Duong et al. [38] leverage frequency-based temporal prefetching candidates to build compact NN-based models, while Zhang et al. [39] propose table-based approximations for NN prefetchers to enhance practicality. In contrast, this work focuses on modeling and achieving higher prefetching performance through NN models and domain knowledge, leaving the optimization of deployment practicality for future work.

In this work, ML plays both indirect and direct roles. We present a comprehensive prefetching framework where multiple ML models collaborate to aid in the selection of prefetchers and to predict future memory accesses.

B. Domain Specific Machine Learning

Domain Specific Machine Learning (DSML) [40] refers to a methodology that integrates domain knowledge into the design, training, and optimization of ML models for domain specific problems. Murdock et al. demonstrate the effectiveness of domain knowledge in improving ML model predictive ability [41]. Domain knowledge have been integrated into ML models in various areas, such as climate modeling [42], turbulence modeling [43], and earth science [44]. There are also several notions related to DSML. Theory-guided data science [45] proposes a new paradigm that is gaining prominence in scientific disciplines when designing data science models. Informed machine learning [46] integrates prior knowledge into the machine learning pipeline by using it as an independent input source. Physics-based machine learning [47] incorporates physical properties into model training by introducing physics-guided loss functions and initialization.

In this work, we introduce a methodology to develop DSML models for a domain specific problem: prefetching for graph applications. While some existing ML-based prefetchers have incorporated some domain features into their modeling, such as program counters [19] and memory configurations [24], we propose a comprehensive approach that considers both the context of the architecture and the context of the computation in developing ML models for prefetching.

C. Prefetching for Graph Applications

Recent work on data prefetching for irregular, indirect access patterns in graph and data-intensive applications has led to diverse approaches. The Indirect Memory Prefetcher (IMP) [48] targets L1 cache prefetching, leveraging a linear correlation assumption in indirect memory accesses, which is particularly effective in graph-based applications. However, its reliance on linearity limits adaptability to complex access patterns. We address this by adopting more flexible ML models for diverse memory accesses. Ainsworth et al. [49] propose CSR-specific prefetching for graph traversal, whereas our method, independent of specific data structures, adapts to diverse application contexts. Prodigy [50], a hardware-software co-design prefetcher, uses compiler support for indirect accesses, adding complexity outside our design scope; in contrast, our approach integrates without additional software. DVR [51] and SVR [52] enhance memory-level parallelism through microarchitectural support, but their microarchitecture-specific optimizations limit their portability across hardware configurations. In contrast, we aim to design a prefetcher that can seamlessly function across different architectures without needing such hardware-specific adaptations. Gretch [53] focuses on virtual address spaces and relies on the detection of data dependencies in virtual memory. In contrast, our approach is not confined to virtual memory spaces, allowing it to function across broader contexts. DMP [54] presents an L1 cache prefetcher that detects indirect memory access patterns using index streams based on a differential matching method.

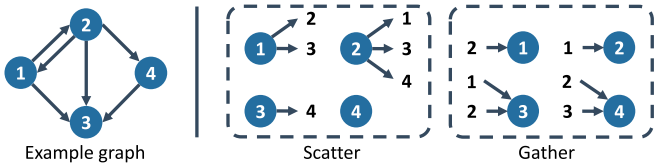


Fig. 1. Scatter and Gather phases in graph analytics.

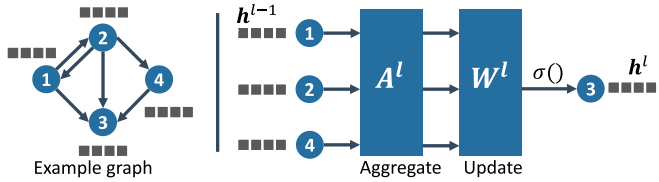


Fig. 2. Aggregate and update phases in graph neural networks.

Algorithm 1: Scatter-Gather Paradigm in Graph Analytics.

```
while not done do
  for all vertex v that need to scatter updates do
    SCATTER(v)
  __synchronize()__
  for all vertex v that have updates do
    GATHER(v)
  __synchronize()__
```

Algorithm 2: Aggregate-Update Paradigm in GNNs.

```
for l = 1...L do
  for vertex v ∈ V^l do
    a_v^l = AGGREGATE(h_u^{l-1} : u ∈ N(v), u ∈ V^{l-1})
    h_v^l = UPDATE(a_v^l, W^l, σ())
```

Our work instead targets multi-core systems, using an ML-based method to learn the interleaved memory accesses in LLC. Existing works such as DMP [54] and SVR [52] require access to the input graph data during execution for prefetching. In contrast, our approach leverages domain knowledge (e.g., the number of phases in the application) and graph metadata (e.g., the number of nodes and edges), without accessing the actual graph data at runtime. In summary, our domain-specific ML-based prefetcher stands out for its adaptability without dependence on data formats, compiler support, or hardware modifications, making it versatile across computational contexts.

III. Background

Our prefetching approach is grounded in the domain knowledge present in graph applications. In this section, we illustrate the widely-used paradigms in graph analytics and graph neural networks that inform our prefetching model development.

A. Graph Analytics

Graph analytics frameworks using various computing paradigms have been developed with exceptional performance [2], [3]. We target graph analytics frameworks with iterative barrier-synchronized phases. For example, *Scatter-Gather* is a widely-used parallel programming paradigm for graph analytics [2], [3]. As is shown in Fig. 1, graph analytics under the Scatter-Gather paradigm operates in two phases:

Scatter: Each node in the graph processes its local data and computes partial results. These partial results are then sent to neighboring nodes in the graph.

Gather: Each node collects the partial results received from its neighbors and combines them to compute the final result.

Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ denotes the set of edges, the computation process of Scatter-Gather is shown in Algorithm 1. For each processing iteration, the two phases alternate, with a barrier synchronizing all nodes between phases.

B. Graph Neural Networks

The Aggregate-Update paradigm [55] is foundational in many GNN architectures [56], [57], [58] and is used to iteratively refine node representations by incorporating information from the surrounding graph structure. Fig. 2 illustrates an example where a graph neural network updates the feature vector of a target node through Aggregation and Updates phases.

Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$, where $\mathcal{V}$, $\mathcal{E}$, and $\boldsymbol{X}$ denote the set of vertices, the set of edges, and the vertex features, and given a GNN model $\mathcal{M}(L, f^l, \boldsymbol{A}^l, \boldsymbol{W}^l)$, where L, f^l, $\boldsymbol{A}^l$ and $\boldsymbol{W}^l$ denote the number of model layers, the hidden dimension in layer l, the graph adjacency matrix in layer l, and the weight matrix in layer l, the GNN model inference process consists of two phases, as shown in Algorithm 2:

Aggregate: Each node aggregates information from its neighboring nodes. The input is the last layer hidden feature $\boldsymbol{h}_v^{l-1}$ for neighboring nodes of v, denoted as $\mathcal{N}(v)$, and the adjacency matrix $\boldsymbol{A}^l$, the output is the aggregated feature $\boldsymbol{a}_v^l$.

Update: Each node updates its own representation based on the aggregated information. The input is the aggregated feature $\boldsymbol{a}_v^l$ and the weight matrix $\boldsymbol{W}^l$, the output is the updated vertex feature for $\boldsymbol{h}_v^l$ layer l, which is typically processed by an activation function Sigmoid $\sigma()$.

Using the domain knowledge of GNN computation, we incorporate the information of phases, input graph structure, and GNN metadata into the development of prefetching models for GNN applications.

IV. Hierarchical Hybrid Prefetching

Memory access patterns vary among different phases of graph processing, with regular and irregular patterns interleaving. To improve the effectiveness of prefetching for various phases and patterns, we propose a novel prefetching framework, Hierarchical Hybrid Prefetching (HHP), which decomposes the memory access patterns and allocate prefetchers.

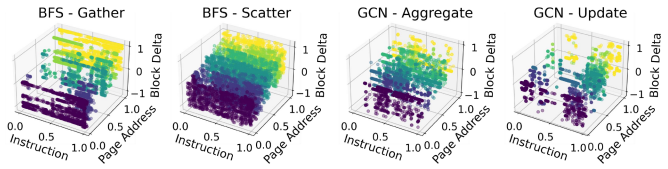


Fig. 3. Pattern distribution varies at different graph processing phases.

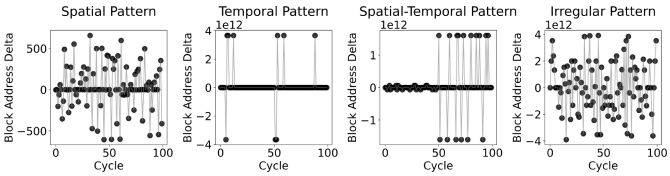


Fig. 4. Various locality patterns exist in memory access sequences.

A. *Diversity in Memory Access Patterns*

We observe diverse memory access patterns in graph applications, which motivates us to decompose these patterns, aiming to optimize the development of ML models for prefetching.

Fig. 3 shows the diverse distributions of memory accesses at different graph processing phases. As examples, we extract traces from the graph analytics application BFS using GPOP framework [2] and the graph neural network algorithm GCN [56] using PyG framework [59]. In Fig. 3, we visualize the trace distribution in three normalized dimensions: instruction index, pages address, and block address delta (the consecutive cache line address difference). The observation that the memory access distribution varies across phases has led us to train specialized ML models for each phase to improve memory access prediction performance.

Fig. 4 provides a granular view of memory access sequences, showcasing typical locality patterns from the aforementioned example applications. These patterns include spatial patterns, where the block address delta is within a small region; temporal patterns, where repeated access of the same distance page occurs; spatial-temporal patterns, which mix spatial and temporal patterns; and irregular patterns, which exhibit limited explicit patterns. These patterns interleave in the memory accesses during application execution. This observation motivates us to distinguish regular simple patterns from complex spatial-temporal and irregular patterns. By focusing on the complex patterns, we aim to train more efficient model for memory access prediction.

B. *Hierarchical Hybrid Prefetching Framework*

We introduce a novel generic framework called Hierarchical Hybrid Prefetching (HHP) to address the diverse distribution and various patterns encountered in graph applications. The overall structure of HHP is illustrated in Fig. 5. We design four hierarchical levels to process the input memory access sequence and use a hybrid prefetching controller to manage the cooperation of multiple prefetchers.

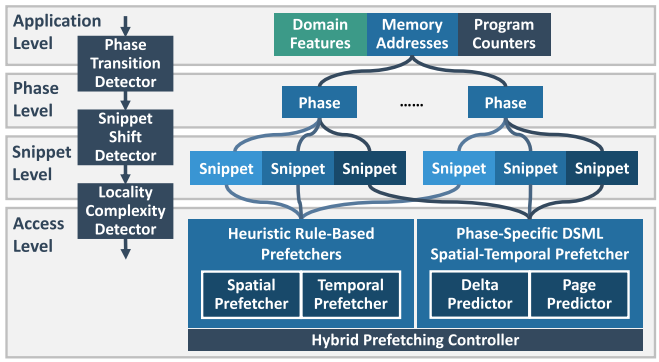


Fig. 5. Overview of the Hierarchical Hybrid Prefetching (HHP) framework.

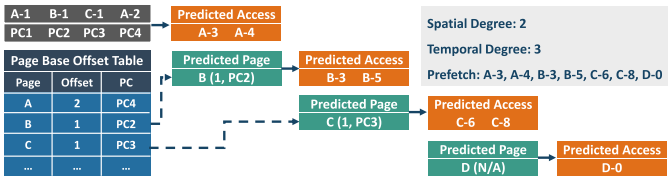


Fig. 6. Chain spatial-temporal prefetching.

Application level: At this level, the HHP framework receives input that comprises memory access addresses, program counters, and domain features. The domain features encompass various aspects, including the application paradigm, characteristics of the input graph structure, and the metadata of GNN models. By incorporating this diverse set of inputs, the HHP framework gains a comprehensive understanding of the memory access behavior within the graph application context.

Phase level: Taking the input from the application level, a *phase transition detector* determines the current graph processing phase. The number of phases is determined by the programming paradigm of the application and we assume access to this domain knowledge before model training. There are two scenarios for phase detection. If online interfaces for software hints are available, phases can be directly allocated based on these hints [60]. Alternatively, if no online interfaces exist, we can offline train a classifier, such as a decision tree [61], using memory access traces extracted from each phase. For each phase, we train phase-specific DSML models for memory access prediction.

Snippet level: We define a snippet as a short piece of sequences sharing similar patterns. To detect a snippet, we propose using an unsupervised *snippet shift detector* to sense the shifts in memory access patterns. We propose using a *locality complexity detector* to assess the complexity of the initial sequence of a snippet. Our approach runs both rule-based and ML-based prefetchers in parallel. Based on the detected complexity for an entire snippet, we select which prefetcher's predictions to use. This selection targets only the predictions, not the active prefetcher, and relies on a straightforward condition derived from the complexity detection, ensuring no additional switching overhead is introduced.

Access level: Memory access prediction and data prefetching happen at the memory access level. We use spatial-temporal prefetching considering the complex patterns in graph applications. Rule-based prefetching combines spatial and temporal prefetchers, while DSML-based prefetching uses a page predictor for temporal page jumps and a delta predictor for spatial predictions within a page.

C. Hybrid Prefetching Controller

We designed a hybrid prefetching controller to manage the prefetcher switching and to handle predictions from multiple ML prediction models.

Prefetcher switching: When a new snippet is detected, the initial short sequence is analyzed for locality complexity. If the sequence is detected as high complexity, the controller allocates phase-specific DSML-based models for prefetching. Prefetcher switching occurs only at the snippet level.

Rule-based prefetchers: Rule-based prefetchers keep operating in parallel and learning access patterns. Predictions from rule-based prefetchers are issued during low-complexity snippets and during the processing of the locality complexity detector. The temporal prefetcher is prioritized over the spatial prefetcher when it can offer predictions, as it demonstrates high accuracy by predicting under exact matches only.

DSML predictors: We propose a Chain Spatial-Temporal Prefetching (CSTP) strategy to manage predictions from the delta predictor and the page predictor, as shown in Fig. 6. Given an input sequence of memory block addresses ("Page-offset", e.g., A-1) and a PC sequence, the two predictors operate in parallel. A Page Base Offset Table (PBOT) records the latest offset and PC for past pages. For a predicted page, the latest offset and PC can be retrieved from the PBOT for further spatial and temporal inference. This chain process continues until either the temporal degree is exceeded or the page offset is missing in the PBOT. Given a spatial degree D_s and temporal degree D_t, the total prefetch degree D_p range is $[D_s + 1, D_s(D_t + 1)]$.

V. Domain Specific Machine Learning for Data Prefetching

With the Hierarchical Hybrid Prefetching framework established, we develop machine learning models as components (detectors and predictors) within this framework. To create high-performance specialized models for prefetching graph applications, we introduce a methodology of Domain Specific Machine Learning (DSML) for data prefetching. This integrates domain knowledge into ML-based prefetching.

A. Overview of DSML for Prefetching

Fig. 7 illustrates the methodology for developing DSML models for data prefetching. By exploiting domain features from the context of the architecture and the computation of a specific application, we can integrate domain knowledge into ML models for prefetching.

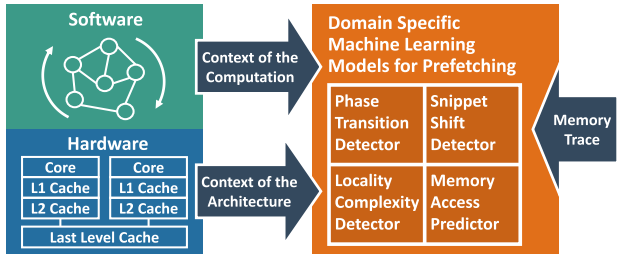


Fig. 7. Methodology for developing DSML models for data prefetching.

B. Domain Features for Prefetching

Knowing the hardware platform of an application, we can integrate domain features from the context of the architecture. Example features include:

- *Platform:* the target architecture on which the computation is executed, such as multi-core platform, multi-CPU cluster, heterogeneous architecture, etc.
- *Memory hierarchy:* the parameters for each level of cache, main memory, flash memory, etc.
- *Memory address configuration:* the bit length of a memory address, the cache line size, the page size, etc.

With the domain knowledge of the running software, we can extract domain features from the context of the computation. Example features include:

- *Computing paradigm:* a style of computation process, such as *Scatter-Gather* [2], [3] in graph analytics and *Aggregate-Update* [55] in graph neural networks.
- *Phase:* a step or a super-step in a computing paradigm. Different phases encompass memory accesses with diverse patterns.
- *Modality:* a mode, a set of features, to describe or represent the computation. To describe the memory accesses, modalities include program counters, memory addresses, memory page sequences, thread IDs, etc.
- *Locality:* the data access patterns in a computation, e.g., the spatial locality and the temporal locality.
- *Thread:* a unit of execution within a process. Interleaved memory accesses in multi-threaded applications make the prediction harder than in single-threaded applications.
- *Coordination:* the way how multiple cores or nodes of a system work together in a parallel computing application, including synchronous and asynchronous coordination.
- *Data representation:* graph data can be represented as an edge list, adjacency list, adjacency matrix, and various sparse graph representations [62].
- *Metadata:* metadata of input graph and application, such as the number of edges and nodes in the input graph of a graph analytics application, and the number of layers and dimensions in a GNN application.

Note that the above describes various features that can be used to define a domain specific model in general. In this work, we leverage domain knowledge, specifically phase, modality, and locality, to guide model design and achieve higher prediction performance using a small set of simple input features. These

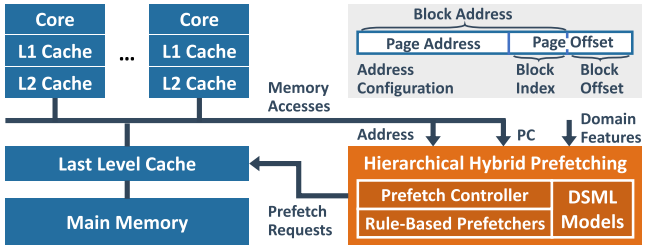


Fig. 8. Multi-core shared memory architecture with hierarchical hybrid prefetching and DSML models for LLC prefetching.

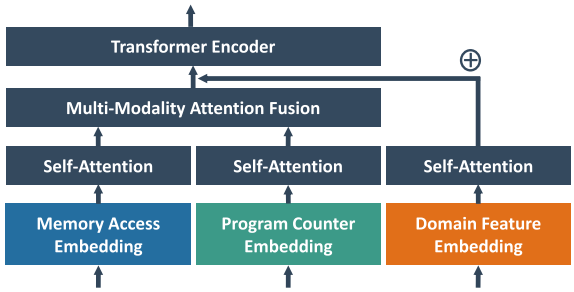


Fig. 9. DAMMA incorporates three types of input embeddings and extract features through three types of attention-based layers.

include the total number of nodes and edges in the input graph, as well as the number of layers and hidden dimensions in the GNN model, as discussed in Section VI-D.

C. DSML for Graph Application Prefetching

Graph analytics and graph neural networks (GNNs) are major application areas, driving progress in fields such as social network analysis [63] and scientific computing [64]. By focusing on optimizations for data prefetching in graph applications, this paper illustrates the role of domain specific modeling in a broad application domain.

We specify optimizations for developing ML models for graph application prefetching by incorporating the context information of the architecture and the computation.

1) Context of Architecture: Our target platform is multi-core shared memory architecture, as shown in Fig. 8. The memory hierarchy presents multi-core processors with private L1 caches (including data cache L1D and instruction cache L1I), private L2 caches, a shared last level cache (LLC), and a shared main memory. We focus on the shared LLC prefetching. The HHP framework with DSML models takes input from the history of LLC requests, predicts future memory accesses, and requests prefetches to LLC. We take into account the memory address configuration of the system. The prefetching is at the block level, leading us to train a model to predict the block index.

2) Context of Computation: In developing DSML models, we integrate domain features from the context of the computation including *modality*, and *structure*, *phase*, and *locality*.

Incorporation of modality: When multi-threaded applications are executed in parallel, it leads to interleaved instructions and a significant irregularity in memory accesses. Memory address sequences alone cannot fully capture this complexity. Instead, considering the instruction perspective, Program Counters (PCs) indicate the execution of a specific thread. We assume PCs are accessible in LLC prefetching, following existing works such as TransFetch [24] and Voyager [19]. Furthermore, we emphasize the PC sequence as an equally vital input modality alongside the address sequence.

Incorporation of structure: For graph analytics, we incorporate the input graph structure, including the number of nodes, the number of edges, and the graph density. For graph neural network, we also incorporate the GNN model metadata, including the neural network layers and the hidden dimensions. These features are normalized and processed as input to the memory access prediction models.

Incorporation of phase: As analyzed in Section IV-A, memory access distribution varies for different phases. In line with the HHP framework, we train phase-specific ML models based on the computing paradigm and the number of phases. We employ a phase transition detector to determine and select the corresponding model for each phase.

Incorporation of locality: We observe wide-range memory access jumps across pages from Fig. 4. Therefore, in addition to predicting block indexes based on spatial locality, we propose predicting memory access pages based on temporal locality. In addition, we measure the complexity of a memory access sequence and train the ML models only using high-complexity patterns that show less regular locality in it, aiming to improve model performance on complex patterns.

D. DAMMA

We propose DAMMA, a Domain Specific Attention-Based Multi-Modality Network Using Attention Fusion, as shown in Fig. 9. DAMMA is a versatile backbone network capable of fulfilling diverse prefetching tasks, including memory access delta prediction and memory page prediction. DAMMA learns embeddings for the input of memory accesses, program counters, and domain features, denoted as E_M, E_P, and E_D. The embeddings are processed using layers based on the attention mechanism [23], including a self-attention layer, a multi-modality attention fusion layer, and a Transformer encoder.

Self-Attention Layer takes an embedding as input, converts them to three matrices through linear projection, then feeds them into a scaled dot-product attention defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where Q represents the queries, K the keys, V the values, and d the dimension of layer input.

Multi-Modality Attention Fusion Layer merges the input of two modalities for memory access E_M and program counters E_P through concatenation and self-attention:

$$E_{cx} = \text{Concat}\,(E_M, E_P)$$

$$\mathrm{MMAF}(E_{cx}) = \mathrm{Attention}\left(E_{cx}W_i^Q, E_{cx}W_i^K, E_{cx}W_i^V\right) \tag{2}$$

Transformer Encoder consists of a Multi-head Self Attention (MSA) and a point-wise feed-forward network (FFN).

$$\mathrm{MSA}(Q,K,V) = \mathrm{Concat}\left(\mathrm{head}_1,\ldots,\mathrm{head}_\mathrm{H}\right)W^O$$

$$\mathrm{head_i} = \mathrm{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{3}$$

$$\mathrm{FFN}(x) = \max\left(0, xW_1 + b_1\right)W_2 + b_2 \tag{4}$$

where $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d\times d}$ and H is the number of heads. In DAMMA, the Transformer encoder takes the point-wise sum between the MMAF and the self-attention of E_D as input.

VI. GraFetch

As an exemplar of the proposed Hierarchical Hybrid Prefetching (HHP) framework and the Domain Specific Machine Learning (DSML) methodology for prefetching, we develop GraFetch to accelerate graph analytics and graph neural network applications through prefetching. We specify our design of DSML models in HHP, including the phase transition detector, snippet shift detector, locality complexity detector, and the memory access predictors.

A. Phase Transition Detector Using Soft Transition

Graph processing phases are defined in software, thus, we use program counters (a.k.a. instruction pointers) to detect the phases. Given a stream of T history program counters at time t, represented as $PC_t = \{pc_{t-T+1},\ldots,pc_{t-1},pc_t\}$, and the corresponding phase as class $n_t \in 0,1,\ldots,N-1$, we can train a classifier as the phase detector that predicts the current phase. A phase transition is detected when two consecutive phase predictions are different.

However, a naive classifier exhibits a notable issue: high false positive rate due to the detector prematurely signaling a phase transition upon predicting a different phase. This encompasses scenarios involving transient pattern fluctuations within a phase and erroneous predictions.

To address this, we propose a *soft transition* mechanism aimed at mitigating this false positive rate, as shown in Algorithm 3. Our approach involves the establishment of a result queue, denoted as Q, which retains past phase inferences. Subsequently, we compare the modes (i.e., the most frequently occurring element) between the initial and latter halves of Q. A transition detection is reported when the two modes are different, effectively mitigating the impact of abrupt pattern shifts. We apply soft transition detection to Decision Tree (DT) [61] and Random Forest for phase detection [65].

Algorithm 3: Soft Transition for Phase Detection.

```
Initialize transition ← False
Initialize history result queue Q and its length limit H
n'_t ← CLASSIFIER(PC_t)                          ▷Inference
Push n'_t to Q
if length(Q) ≥ W then                            ▷Soft transition
    n'_h ← mode(Q[0 : H/2])
    n'_r ← mode(Q[H/2 : H])
    if n'_h ! = n'_r then
        transition ← True
        Reset Q
```

B. Snippet Shift Detector Using Soft-KSWIN

Snippet detecting is critical for assigning specific models and performing hybrid prefetching. We use unsupervised learning to detect snippet shifts. The distribution of memory access block address sequence X_t is defined as $P(X_t)$. If a snippet shift occurs at time t, then:

$$P(X_t) \neq P\left(X_{t-1}\right) \tag{5}$$

The phase transition is described as a concept drift [66] in the context of machine learning. Kolmogorov–Smirnov Windowing (KSWIN) [67] is a state-of-the-art concept drift detection approach.

Kolmogorov–Smirnov Windowing (KSWIN): KSWIN is based on the two sample Kolmogorov–Smirnov (K-S) test [68], which estimates the probability that two sets of samples were drawn from the same distribution. The K-S statistic D is the absolute distance between the two empirical cumulative distributions F_A and F_B:

$$D_{A,B} = \sup_x\left(|F_A(x) - F_B(x)|\right) \tag{6}$$

where $F_{(\cdot)}(x) = \frac{1}{n}\sum_{i=1}^{n} I_{[-\infty,x]}(X_i)$ is the empirical distribution function for n ordered observations X_i; $I_{[-\infty,x]}(X_i)$ is an indicator function that equals to 1 if $X_i < x$ and 0 otherwise; $\sup(x)$ is the supremum of the set of distances. KSWIN detects pattern shifts by comparing the distributions of two windows: one containing h history samples H and the other containing r recent samples R. These windows are sampled from a sliding window Ψ, which keeps w most recent points from the stream, as shown below:

$$R = \{\mathbf{x}_i \in \Psi\}_{i=w-r+1}^{w} \tag{7}$$

$$H = \{\mathbf{x}_i \in \Psi \mid i \leq w-r; p(\mathbf{x}) = \mathcal{U}\left(\mathbf{x}_i \mid 1, w-r\right)\} \tag{8}$$

We can reject the null hypothesis (that there is no statistically significant difference between the two observations) at a significance level of α if the following inequality is satisfied:

$$D_{H,R} > \sqrt{-\ln\left(\frac{\alpha}{2}\right)\frac{1+\frac{r}{h}}{2r}} \overset{(h=r)}{=} \sqrt{-\ln\left(\frac{\alpha}{2}\right)\frac{1}{r}} \tag{9}$$

KSWIN reports a snippet shift when D exceeds the threshold. This "hard" detection process ignores the fluctuation when the pattern is shifting, leading to *unstable* predictions during memory access pattern shifts, as shown in Fig. 10.

Soft-KSWIN: We propose Soft-KSWIN, a domain specific variant of KSWIN that uses the soft detection mechanism (Section VI-A) to avoid the instability of KSWIN. We introduce a soft history window H' sampled from a dynamic set of history

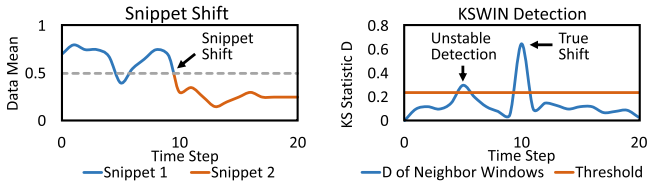


Fig. 10. Example of KSWIN unstable performance in snippet shift detection.

Algorithm 4: Soft-KSWIN for Snippet Shift Detection.

1: Initialize α, th_r
2: Initialize sliding window Ψ and its length limit w
3: Initialize soft history window H' with size h
4: Initialize recent window R with size r
5: Initialize $counter \leftarrow 0$, $detection \leftarrow 0$
6: Initialize $shift \leftarrow False$
7: Initialize $threshold \leftarrow \sqrt{-\ln(\frac{\alpha}{2})\frac{1+\frac{r}{h}}{2r}}$
8: **if** length(Ψ) $\geq w$ **then**
9: Pop the oldest data point from Ψ
10: Push the new data point to Ψ
11: $counter \leftarrow counter + 1$
12: $H' \leftarrow$ Sample uniformly from Ψ[1:w-r-counter]
13: $D_{H',R} \leftarrow \sup(|F_{H'}(x) - F_R(x)|)$ ▷Inference
14: **if** $D_{H',R} > threshold$ **then** ▷Soft detection
15: $detection \leftarrow detection + 1$
16: **if** $counter \geq r$ **then**
17: **if** $detection/counter > th\_r$ **then**
18: $shift \leftarrow True$
19: Reset the model
20: **else**
21: Push new data point to Ψ

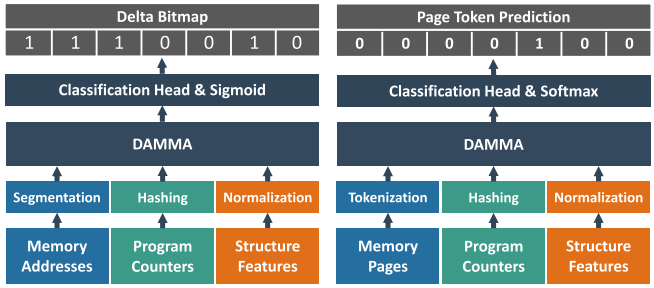
(a) KSWIN sampling. (b) Soft-KSWIN sampling.

Fig. 11. Comparison between KSWIN and Soft-KSWIN. KSWIN samples history data from a fixed-size history window while Soft-KSWIN shrinks the history window boundary to ensure the history data unpolluted by new pattern.

data points, as shown in Fig. 11 and (10):

$$H' = \{\mathbf{x}_i \in \Psi \mid i \leq w - r - c; p(\mathbf{x}) = \mathcal{U}(\mathbf{x}_i \mid 1, w - r - c)\} \tag{10}$$

where c is a counter initiated when a positive detection occurs. The Soft-KSWIN is shown in Algorithm 4. The history window samples from the sequence not polluted by the new pattern. As data flows through the stream, if snippet shift detection occurs, both the counter and the number of detections are incremented (lines 11-15). When an entirely new recent window is sampled and $counter \geq r$, if the ratio between detections and the counter is larger than a threshold th_r, a final detection is declared to be positive and the model is reset for future detections (lines 16-21).



(a) Spatial delta predictor. (b) Temporal page predictor.

Fig. 12. DAMMA-based models for delta and page predictions.

C. Locality Complexity Detector Using Composite Entropy

We employ a statistical approach to quantify the complexity of a memory access sequence. We leverage Shannon's entropy:

$$\mathrm{H}(X) = -\sum_{x \in X} p(x) \log p(x) \tag{11}$$

where X denotes the sequence, and x represents the distinct discrete values within X. For each memory access sequence of length N, we analyze two distinct sequences:

Page sequence X_p: The page sequence indicates the temporal locality. Repeated access of a page is easier for prediction though they are in far distance. More distinct pages in a sequence indicate low temporal locality and difficult to predict.

Block address delta sequence X_d: The delta sequence indicates the spatial locality. A typical spatial pattern shows nearby accesses with the same stride, leading to a lower number of distinct block address deltas.

We compute entropy for both X_p and X_d at length N, and subsequently merge the two entropies using their geometric mean. The resulting value is then normalized by dividing it by $logN$. The composite entropy H_c is defined in (12):

$$H_c = \sqrt{H(X_p)H(X_d)} / \log N \tag{12}$$

By setting a locality complexity threshold $\tau \in [0, 1]$, we define low-complexity patterns for $H_c \leq \tau$ and high-complexity patterns otherwise. We train the memory access predictors using only the sequences with complex patterns. In this way, the trained predictors avoid bias from the simple patterns. At runtime, we allocate heuristic rule-based prefetchers for low-complexity sequences and allocate the DSML models for prefetching high-complexity sequences.

D. Phase-Specific DSML Spatial-Temporal Prefetcher

We train phase-specific predictors based on DAMMA. To handle complex patterns, we design a spatial delta predictor and a temporal page predictor, as shown in Fig. 12.

1) Input Preprocessing: We use memory addresses, program counters, and domain features for our predictor input.

Memory addresses: Following the address segmentation approach [24], we split a block address into $S = \lceil \frac{p}{c} \rceil + 1$ segments for a p-bit page address and c-bit block index.

TABLE I
GRAPH FRAMEWORKS AND ALGORITHMS FOR EVALUATION

Domain	Framework	Paradigm	Applications
GA	GPOP [2]	Scatter-Gather	BFS, CC, PR, SSSP
	X-Stream [3]	Scatter-Gather	BFS, CC, PR, SSSP
GNN	PyG [6]	Aggregate-Update	GCN, GAT, GIN, SAGE
	DGL [7]	Aggregate-Update	GCN, GAT, GIN, SAGE

TABLE II
GRAPH NEURAL NETWORK DATASETS

Dataset	Vertices	Edges	Features	Classes
Cora (CO)	2708	10556	1433	7
Flickr (FL)	89,250	899,756	500	7
Reddit (RE)	232,965	116,069,919	602	41
Yelp (YE)	716,847	6,977,410	300	100
AmazonProduct (AP)	1,569,960	264,339,468	200	107

TABLE III
SIMULATION PARAMETERS

Parameter	Value
CPU	4 GHz, 4 cores, 4-wide OoO, 256-entry ROB, 64-entry LSQ
L1 I-cache	64 KB, 4-way, 4-cycle latency
L1 D-cache	64 KB, 4-way, 4-cycle latency
L2 Cache	512 KB, 8-way, 10-cycle latency
LL Cache	2 MB, 16-way, 20-cycle latency
DRAM	$t_{RP} = t_{RCD} = t_{CAS} = 12.5$ ns, 2 channels, 8 ranks, 8 banks, 32K rows, 8GB/s bandwidth

TABLE IV
PRECISION AND RECALL OF PHASE TRANSITION DETECTION

	Size†	GPOP		X-Stream		PyG		DGL	
Model	(KB)	P	R	P	R	P	R	P	R
DT	1.929	0.328	1.000	0.494	1.000	0.399	1.0000	0.317	1.000
RF	63.71	0.661	1.000	0.778	1.000	0.665	1.000	0.623	1.000
Soft-DT	1.929	0.925	1.000	1.000	1.000	0.969	1.000	0.975	1.000
Soft-RF	63.71	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

† Model size averaged across all applications.

Memory pages: We tokenize the memory page address and use the token index for ML model input. The vocabulary of page is in thousands which can be directly processed [19].

Program counters: We use a H-bit length folding method as the hash function to compress the PC value. The hashed value is divided by 2^H for normalization.

Structure features: We incorporate data structure features of graph analytics and GNN applications as the domain features.

- For the input graph, we extract the graph structure information including: number of nodes, number of edges, density, diameter, and the average degree. We scale the features and concatenate them as an input vector.
- For GNN applications, we also incorporate the GNN model metadata, including: the number of layers, dimension of input features, dimension of node embedding, GNN layer type, and the aggregation type. The numeric values of layer and dimension sizes are scaled while the types are encoded using one-hot.

2) Spatial Delta Predictor: Fig. 12(a) illustrates the model of the spatial delta predictor.

Input: It uses the memory address address sequence, PC sequence, and domain features as inputs.

Output: The model predicts multiple future deltas within a spatial range, i.e., a page size. The sum of the current address and the predicted delta is the prediction of future accesses.

Training: The model is trained using labels from future deltas in the form of a bitmap for multi-label classification. We use binary cross entropy as the loss function.

3) Temporal Page Predictor: Fig. 12(b) illustrates the model of the temporal page predictor.

Input: It take memory access page sequence to replace the memory address sequence as input.

Output: The model outputs the probability of the next future page as determined by the Softmax function.

Training: The model is trained using the future page token as label. We use categorical cross entropy as the loss function.

VII. EVALUATION

A. Experimental Setup

1) Benchmarks: Table I shows the benchmark frameworks and applications we use for evaluation.

Graph Analytics (GA): We use two popular graph processing frameworks: GPOP [2] and X-Stream [3]. We use the built-in applications of the frameworks for evaluation, including Breadth-First Search (BFS), Connected Components (CC), PageRank (PR), and Single Source Shortest Path (SSSP).

Graph Neural Networks (GNN): We use two popular GNN frameworks PyG [6] and DGL [7]. We implement four GNN models using the two frameworks, including Graph Convolutional Networks (GCN) [56], Graph Attention Networks (GAT) [57], Graph Isomorphism Network (GIN) [69], and GraphSAGE (SAGE) [58].

2) Datasets: We use popular graph datasets for our prefetching experiments, as is shown in Table II.

3) Simulator: We use ChampSim [70] for memory access traces generation and prefetcher evaluation. We use the same simulator parameter as in state-of-the-art prefetchers [19], [24], as shown in Table III.

4) Training: We train a set of DSML models for a single application using all the datasets. We use the first iteration of the graph applications to train the models and use the subsequent iteration of the application execution for evaluation.

B. Evaluation of Phase Transition Detection

1) Metrics: We use Precision (P) and Recall (R) [71] to evaluate the prediction of phase transitions. A transition is counted when the prediction of phase changes.

2) Results: We implement Decision Tree (DT) and Random Forest (RF) for phase transition detection, incorporating the soft transition mechanism (Soft-DT and Soft-RF). Table IV shows their average performance. While all detectors achieve recall = 1, naive DT and RF exhibit low precision due to false positives.

TABLE V
STATISTICS OF SNIPPETS DETECTED USING SOFT-KSWIN

Statistics	GPOP	X-Stream	PyG	DGL
Count of Snippets	633-1463	212-291	5424-5482	1789-5184
Length of Snippets	173-354	181-246	279-830	141-968

TABLE VI
DAMMA MODEL CONFIGURATION

Configuration	Value	Configuration	Value
History N	9	Transformer heads	4
Attention dimension	64	Transformer layer	1
Fusion dimension	128	MLP dimension	128
Transformer dimension	128	MLP head layer	1

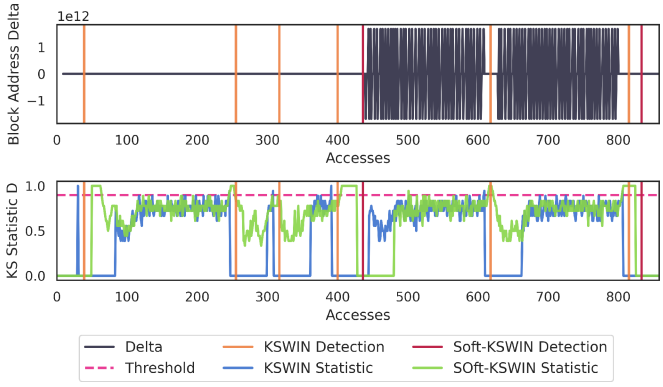


Fig. 13. Case study on snippet detection of KSWIN and Soft-KSWIN.

For average precision, soft transition improves DT from 0.384 to 0.967 and improves RF from 0.682 to 1.

C. Evaluation of Soft-KSWIN for Snippet Detection

Table V presents the statistical results of snippet detection using Soft-KSWIN. The results show that GNN applications contain more snippets (up to 5482) compared to GA applications (up to 1463). Additionally, GNN applications exhibit longer maximum sequence lengths (up to 968 memory accesses) than GA applications (up to 354 memory accesses). These results suggest that GNN applications have longer traces, more shifts among prefetchers in an iteration, and longer stable patterns within a phase.

Fig. 13 illustrates a case study of how Soft-KSWIN stabilizes snippet detection performance. The sequence shows memory access deltas from PageRank using GPOP along with KS statistics of detectors. While KSWIN rapidly detects phases when KS statistic D exceeds a threshold, it can trigger multiple detections due to abrupt shifts. Raising the threshold risks missing true transitions. Soft-KSWIN avoids these unstable detections with only a small lag.

D. Evaluation of Composite Entropy for Complexity Detection

Fig. 14 shows the proportion of low-complexity and high-complexity snippets under threshold $\tau = 0.2, 0.4, 0.6$, and *Mean:*

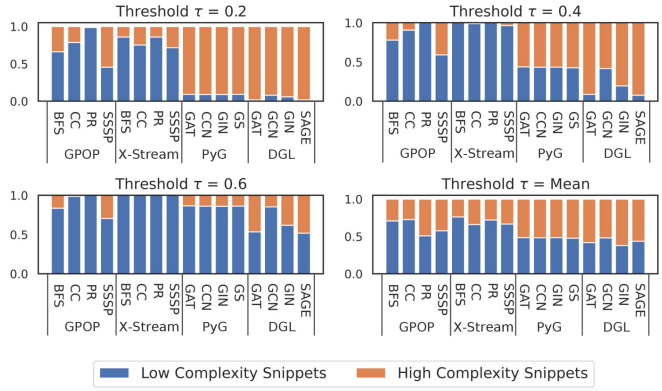


Fig. 14. Proportion of low-complexity and high-complexity snippets given various thresholds in composite entropy. We also employ the mean composite entropy of the training set as an adaptive threshold for each application.

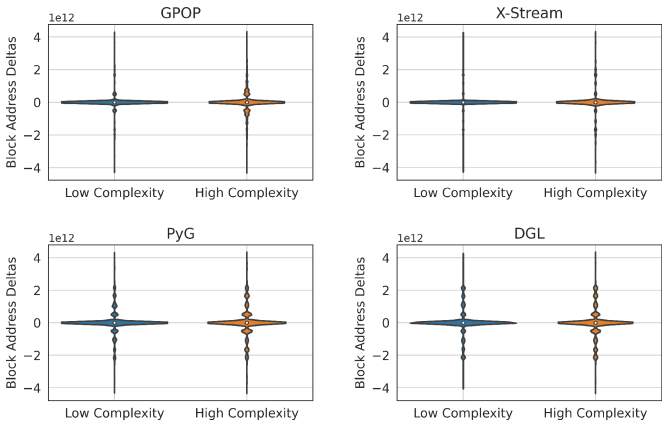


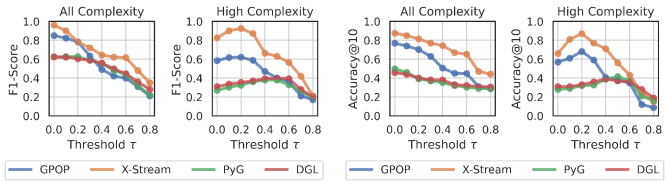
Fig. 15. Distribution of memory access patterns under low-complexity and high-complexity snippets. We use block address deltas to represent the memory access patterns and use mean threshold to classify the complexity.

the mean composite entropy of the application. For threshold $\tau = 0.2$, the average proportions of low complexity snippets for GPOP, X-Stream, PyGs, and DGL are 72.3%, 79.7%, 9.1%, and 4.4%. This clearly shows the distribution of regular and irregular memory access patterns. GA applications GPOP and X-Stream present more regular patterns while GNN applications show more irregular complex patterns. While increasing the threshold value to 0.4 and 0.6, more snippets are classified as low-complexity. This trend also indicates that memory accesses are diverse and have high variance. To balance the workload between rule-based prefetchers and ML model training, we use a mean threshold in our experiments. This mean threshold helps extract high complexity snippets from both training and testing sets for evaluating DSML memory access predictors.

We investigate the distribution of memory access patterns by analyzing block address deltas (deltas) under low-complexity and high-complexity snippets, as shown in Fig. 15. We use the mean threshold method of composite entropy to classify the complexity. Fig. 15 shows that low-complexity snippets exhibit more consistent and localized memory accesses, as evidenced

TABLE VII
EFFECTIVENESS OF PC AS MODALITY

Task	Model	GPOP	X-Stream	PyG	DGL	Mean
Delta Prediction (F1-Score)	Base	0.5965	0.8786	0.3425	0.3236	0.5353
	+ PC (Context)	0.5968	0.8793	0.3427	0.3236	0.5356
	+ PC (Modality)	**0.5994**	**0.8862**	**0.3439**	**0.3296**	0.5398
Page Prediction (Acc@10)	Base	0.5477	0.7983	0.3701	0.3369	0.5133
	+ PC (Context)	0.5479	0.7992	0.3701	0.3372	0.5136
	+ PC (Modality)	**0.5742**	**0.8143**	**0.3704**	**0.4709**	**0.5574**



(a) τ impact to delta prediction. (b) τ impact to page prediction.

Fig. 16. Impact of locality complexity threshold to model performance.

by the narrow and concentrated distribution around the median, indicating high spatial locality and predictable memory access behavior. In contrast, high-complexity snippets show a broader distribution, reflecting greater variability and less predictability in memory accesses. In addition, we observe that GNN workloads (PyG and DGL) show a greater proportion of larger deltas and more outliers compared with GA workloads (GPOP and X-Stream). These findings underscore the need for tailored optimization strategies to address the challenges posed by high-complexity snippets.

E. Evaluation of DAMMA-Based Predictors

1) Metrics: We use F1-Score to evaluate the performance of spatial delta prediction. The F1-Score measures the model performance in accurately identifying prefetch candidates. We use accuracy-at-10 (accuracy@10) to evaluate page prediction. This metric is chosen because a predicted page may not be accessed immediately but could still benefit near-future memory requests if it falls within this window. By considering a window of 10 future accesses, we align with established practices in the field, as seen in prior work [22].

2) Ablation Study: We evaluate the performance of the predictors using the high-complexity snippets detected using the mean threshold. We evaluate our approach by adding components starting from a base model:

Base model (B): an attention-based model taking memory access address sequence as input, trained with all snippets.

Modality of PC (M): adding the program counters as a modality of input to the base model.

Structure features (S): adding the domain features of graph structure and GNN metadata as the model context input.

Phase specificity (P): building multiple models, each model is trained and tested within one phase specifically.

Locality specificity (L): training a model using only high complexity snippets based on the mean threshold.

The configuration of the eventual DAMMA model is shown in Table VI. Tables VIII and IX present the ablation study results for our spatial delta and temporal page predictors. On average, using PC modality improves F1-score by 0.83% for delta prediction and 2.23% for page prediction. Adding domain features further enhances F1-score by 1.72% for delta prediction and 4.30% for page prediction compared to base models. Phase-specific models outperform general models by 2.60% for delta prediction and 5.56% for page prediction. Notably, models trained on high complexity snippets achieve 2.12% higher F1-score for delta prediction and 2.53% higher for page prediction than those trained on all snippets, due to a focused learning on complex patterns. Overall, optimized models show a 7.4% higher F1-score for delta prediction and 15.35% higher accuracy@10 for page prediction compared to basic models.

3) Effectiveness of PC as Modality: To evaluate the effectiveness of our approach of using PC as a modality, we compare these two methods:

Base + PC (Context): using PC as a context input rather than a modality. PCs are processed using domain feature embeddings and this model does not include the attention fusion layer. This method is used in TransFetch [24].

Base + PC (Modality): using PC as a modality and using a fusion layer to merge the input of address and PC. This is used in DAMMA in this work.

The results of delta and page prediction using different methods of incorporating PCs are shown in Table VII. Using PC as context yields only minimal improvement over the base models. In contrast, using PC as a modality significantly increases performance. Particularly, with page prediction, it shows an 8.53% higher accuracy@10 compared to using PC as context.

4) Impact of Locality Complexity Detection: We train the predictors using high-complexity snippets given by various locality complexity thresholds. We test the predictors on two scenarios: 1) the whole testing set including snippets of both low and high complexity, and 2) only the high complexity snippets given by the mean threshold.

Fig. 16 shows the delta and page prediction under these scenarios. As the threshold τ increases, the training dataset becomes smaller with fewer simple patterns. When testing on snippets with all snippets, the model performance drops rapidly because fewer easy-to-predict patterns are used in training. When testing on high-complexity snippets, the model performance increases first due to a higher proportion of complex patterns in training. When τ increases further, the training set becomes insufficiently large, leading to decreased performance. To balance the training dataset size and the snippet complexity, we take the mean composite entropy of each application as the threshold.

F. Prefetching Evaluation

1) Baselines: We compare GraFetch with state-of-the-art rule-based prefetchers and ML-based prefetchers:

Best-Offset (BO) [12]: rule-based spatial prefetcher that predicts delta patterns within a page.

Irregular Stream Buffer (ISB) [15]: rule-based temporal prefetcher based on record and replay.

TABLE VIII
DELTA PREDICTION F1-SCORE

Model					GPOP				X-Stream				PyG				DGL				Mean
B	M	S	P	L	BFS	CC	PR	SSSP	BFS	CC	PR	SSSP	GCN	GAT	GIN	SAGE	GCN	GAT	GIN	SAGE	
✓					0.3159	0.7916	0.9379	0.3405	0.9095	0.7942	0.9111	0.8997	0.3449	0.3504	0.3413	0.3333	0.3674	0.3036	0.3373	0.2862	0.5353
✓	✓				0.3172	0.8009	0.9354	0.3442	0.9158	0.7988	0.9216	0.9085	0.3462	0.3536	0.3416	0.3340	0.3739	0.3007	0.3510	0.2926	0.5398
✓	✓	✓			0.3277	0.7826	0.9284	0.3502	0.9147	0.7965	0.9195	0.9030	0.3618	0.3674	0.3466	0.3709	0.4026	0.3080	0.3773	0.3276	0.5490
✓	✓	✓	✓		0.3364	0.8218	0.9423	0.3459	**0.9253**	0.8264	**0.9306**	0.9060	0.3680	0.3738	**0.3493**	0.3833	0.4286	0.3411	0.3946	0.3401	0.5633
✓	✓	✓	✓	✓	**0.3469**	**0.8227**	**0.9576**	**0.3544**	0.9233	**0.8448**	0.9243	**0.9120**	**0.3816**	**0.3863**	0.3490	**0.3992**	**0.4502**	**0.3621**	**0.4218**	**0.3686**	**0.5753**

TABLE IX
PAGE PREDICTION ACCURACY@10

Model					GPOP				X-Stream				PyG				DGL				Mean
B	M	S	P	L	BFS	CC	PR	SSSP	BFS	CC	PR	SSSP	GCN	GAT	GIN	SAGE	GCN	GAT	GIN	SAGE	
✓					0.4813	0.4635	0.6472	0.5989	0.8003	0.7822	0.8035	0.8074	0.4035	0.3590	0.3843	0.3337	0.3823	0.3088	0.3319	0.3244	0.5133
✓	✓				0.4821	0.5624	0.6516	0.6010	0.8052	0.7879	0.8400	0.8242	0.4048	0.3620	0.3812	0.3337	0.3855	0.3165	0.3326	0.3244	0.5247
✓	✓	✓			0.5691	0.5799	0.6771	0.6211	0.8774	0.7964	0.8802	0.8480	0.4222	0.3672	0.3822	0.3405	0.4055	0.3289	0.3344	0.3256	0.5472
✓	✓	✓	✓		0.6211	0.6199	0.7337	0.6584	0.8840	0.8043	0.9223	0.8817	0.4582	0.3998	0.3820	0.3770	0.4057	0.3784	0.3669	0.3496	0.5777
✓	✓	✓	✓	✓	**0.6266**	**0.6795**	**0.7550**	**0.6655**	**0.8949**	**0.8135**	**0.9225**	**0.8979**	**0.4736**	**0.4188**	**0.3881**	**0.3819**	**0.4235**	**0.3914**	**0.3820**	**0.3626**	**0.5923**

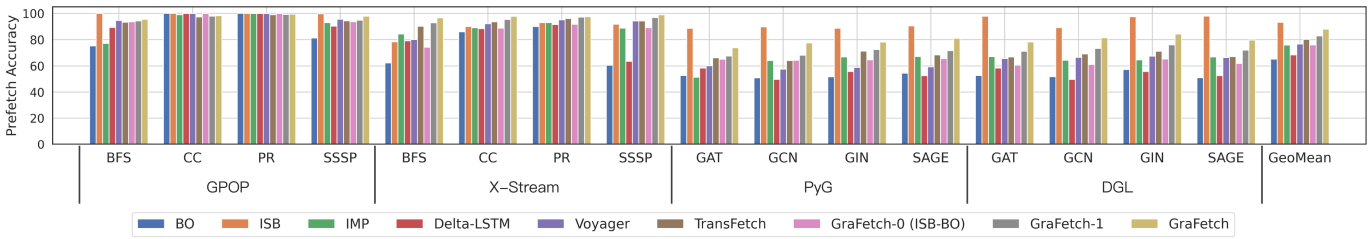


Fig. 17. Prefetch accuracy of GraFetch and the baseline prefetchers.

Indirect Memory Prefetcher (IMP) [48]: rule-based prefetcher detecting indirect memory accesses assuming linear corelation among accesses, widely used in applications such as graph analytics and sparse matrix multiplications.

Delta-LSTM [21]: LSTM-based spatial prefetcher that takes delta as input and predicts the next delta within a page.

Voyager [19]: ML-based prefetcher using two LSTM-based models to predict temporally the next page and offset.

TransFetch [24]: ML-based prefetcher using attention-based model to predict deltas within a spatial range beyond a page.

We train and evaluate all ML-based prefetchers using FP32 precision, consistent with prior studies [19], [21], [24]. This ensures fair comparison and preserves modeling accuracy. For practical hardware deployment, these models can be compressed using techniques such as knowledge distillation [72] and 8-bit quantization of model weights [73]. Exploring reduced-precision optimizations and their impact on performance and accuracy is an important direction for future work.

2) GraFetch Configuration: In the HHP framework, we implement BO as the rule-based spatial prefetcher and ISB as the temporal prefetcher. By setting the locality pattern threshold τ, we evaluate the three variants in GraFetch:

GraFetch: $\tau = Mean$ to enable hybrid usage of the heuristic rule-based prefetchers and the DSML models.

GraFetch-0: $\tau = 0$ to disable the DSML models, equivalent to ISB-BO hybrid prefetching.

GraFetch-1: $\tau = 1$ to disable the rule-based models, equivalent to prefetching using only DAMMA-based models.

3) Metrics: We use prefetch accuracy, prefetch coverage, and IPC improvement [74] to evaluate the prefetchers.

4) Results of Prefetching: Fig. 17 shows the prefetch accuracy for each application. ISB achieves the highest prefetch accuracy at 93.11% on average because it predicts only for exact matches. GraFetch achieves 87.97% accuracy, outperforming the baselines BO, IMP, Delta-LSTM, Voyager, TransFetch, GraFetch-0, and GraFetch-1, which achieve accuracy of 65.12%, 75.87%, 68.42%, 76.63%, 80.26%, 76.016% and 82.88%, respectively. Fig. 18 shows the prefetch coverage results. GraFetch achieves the highest coverage at 38.57%, outperforming the baselines BO, ISB, IMP, Delta-LSTM, Voyager, TransFetch, GraFetch-0, and GraFetch-1, which achieve coverage of 24.38%, 10.72%, 23.47%, 24.45%, 32.16%, 31.88%, 31.11%, and 36.61%. Fig. 19 illustrates the IPC improvement using the implemented prefetchers. GraFetch achieves up to 19.50% IPC improvement for graph analytics and up to 5.68% IPC improvement for GNN, overall 7.22% IPC improvement on average, outperforming the baselines BO, ISB, IMP, Delta-LSTM, Voyager, TransFetch, GraFetch-0,

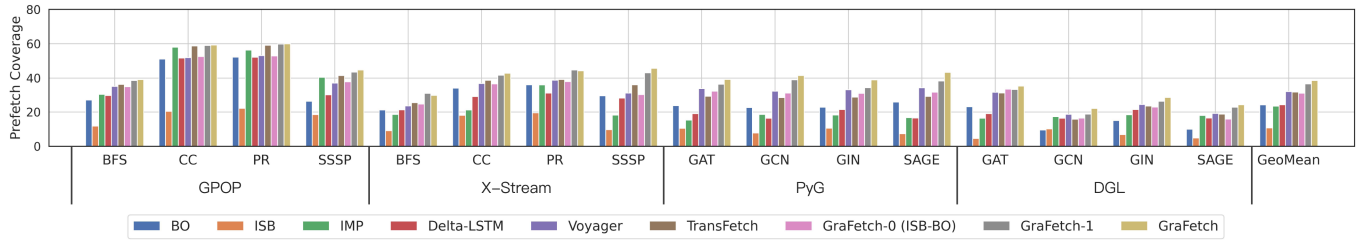


Fig. 18. Prefetch coverage of GraFetch and the baseline prefetchers.

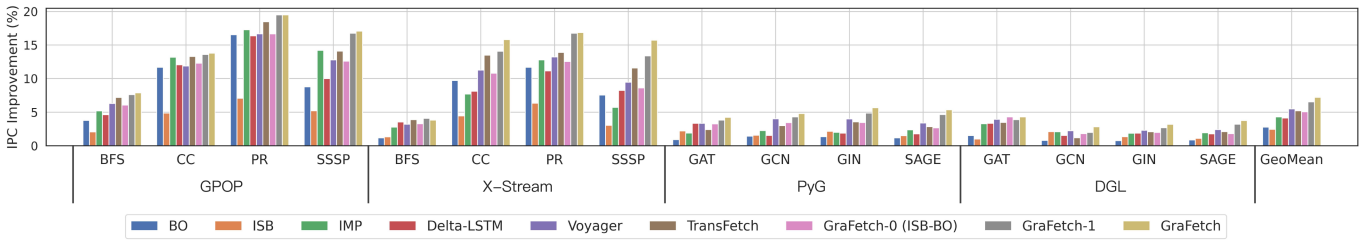


Fig. 19. IPC improvement using GraFetch and the baseline prefetchers.

and GraFetch-1, which achieve IPC improvement at 2.79%, 2.47%, 4.33%, 4.16%, 5.49%, 5.23%, 5.07%, and 6.55%, respectively.

Notably, GraFetch-1 outperforms all ML-based baselines, demonstrating the effectiveness of our DSML-based models for prefetching. GraFetch-0 using only the rule-based prefetchers achieves 5.07% IPC improvement, only 0.15% lower than TransFetch. This indicates that existing ML models learn mainly from simple patterns, which rule-based prefetchers also handle well. By training with complex patterns exclusively and cooperating with rule-based prefetchers, GraFetch provides 2.15% and 0.67% higher IPC improvement compared to GraFetch-0 and GraFetch-1.

We observe that IMP achieves high prefetch coverage for GPOP benchmarks, on average 44.81%, comparable to ML-based prefetcher TransFetch at 47.79%. This is because GPOP uses CSR format input which is a well fit to the learning rule of IMP. However, IMP shows limitations on other graph application frameworks due to the low flexibility of its rule-based method nature. With respect to the graph processing methods, we observe that the GPOP framework that preprocesses graphs into CSR format has shown higher prefetching performance for all prefetchers on average, compared with X-Stream using binary edge list and COO format used in GNN applications.

We also observe different prefetching performance between GA (GPOP and X-Stream) and GNNs (PyG and DGL). For GA, GraFetch results in 12.48% IPC improvement on average, outperforming the best rule-based baseline BO (7.12%) and ML-based baseline TransFetch (10.98%). For GNNs, GraFetch results in 4.18% IPC improvement on average, outperforming the best rule-based baseline ISB (1.10%) and ML-based baseline Voyager (3.14%). This aligns with the snippet complexity discussed in Section VII-D and in Fig. 14. Prefetching for GNN applications is challenging due to their higher proportion of high-complexity memory access patterns, as demonstrated in Fig. 15.

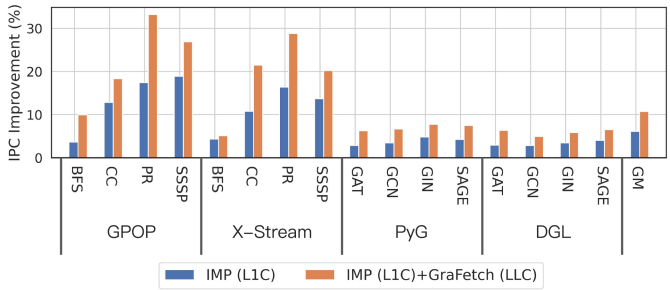


Fig. 20. GraFetch as LLC prefetcher and IMP as L1C prefetcher.

Compared to GA workloads, GNNs exhibit more frequent large-range jumps and a wider distribution of access patterns, resulting in lower prefetching performance. Despite these challenges, our method achieves superior prefetcher performance for GNNs compared to both rule-based and ML-based approaches, which demonstrates the effectiveness of our framework. Our approach using the HHP framework effectively analyzes the pain points of an application, guiding the future optimizations of application development and prefetching strategies.

5) Effectiveness of Hybrid Prefetching: To evaluate the hybrid prefetching strategy driven by composite entropy, we compare GraFetch with the widely used hybrid prefetching technique SBP and the upper-bound performance of an ideal prefetcher.

6) GraFetch With L1 Prefetcher: We further evaluate our prefetching within a system that already incorporates an L1 cache prefetcher (L1C). IMP is designed for L1 cache prefetching by monitoring consecutive processor requests and detecting streaming and indirect memory accesses. The results of IMP as L1C prefetcher and GraFetch as LLC prefetcher are shown in Fig. 20. On average, IMP achieves 6.11% IPC improvement through L1C prefetching, 10.61% for GA, and 3.54% for GNNs. GraFetch provides additional speedup for LLC prefetching,

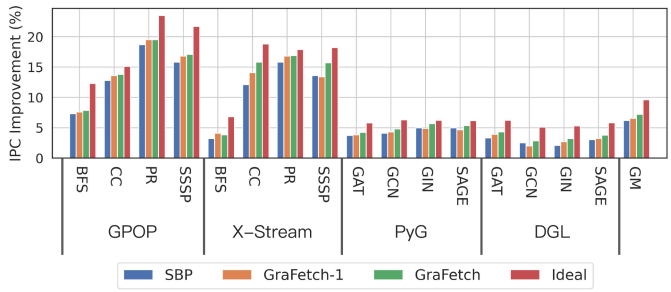


Fig. 21. Comparing GraFetch with hybrid prefetching SBP and ideal prefetching.

TABLE X
COMPUTATIONAL COMPLEXITY OF PREDICTORS IN GRAFETCH AND STATE-OF-THE-ART ML-BASED PREFETCHERS

ML Prefetchers	# Param† (K)	# FLOPs (M)	Critical‡ Path	IPC Impv(%)
Delta-LSTM	346.3	3.13	$\mathcal{O}(nl)$	4.16
Voyager	468.4	11.28	$\mathcal{O}(nl)$	5.49
TransFetch	432.5	3.38	$\mathcal{O}(l)$	5.23
GraFetch	903.1*	7.76	$\mathcal{O}(l)$	7.22

† Parameters in the neural network-based predictors.
‡ n is the sequence length; l is the number of layers.
∗ GraFetch includes delta and page predictors across two phases.

leading to 10.72% IPC improvement on average, 17.87% for GA, and 6.42% for GNNs.

The Sandbox Prefetcher (SBP) [75]: SBP monitors the accuracy of multiple offset prefetchers at runtime using a Bloom filter and selects the highest one after a period of executing, similar in concept to the tournament branch predictor [76]. We implement an enhanced SBP variant, extending its adaptability to include all types of prefetchers. We use BO, ISB, and DAMMA-based models (GraFetch-1) as the input prefetchers.

Ideal prefetcher: An ideal prefetcher prefetches in LLC with perfect precision and zero latency, showing the upper bound of LLC prefetching performance.

The results are shown in Fig. 21. On average, SBP, GraFetch-1, GraFetch, and ideal prefetchers achieve 6.19%, 6.55%, 7.22%, and 9.63% IPC improvement. While for some cases SBP effectively improves the performance of GraFetch-1 through hybrid prefetching (such as PyG GIN and X-Stream SSSP), in most cases, SBP fails and makes negative optimization. This is because the greedy strategy of SBP leads to the limitation of response lag, a sub-optimal prefetcher works for a period until the average performance of another prefetcher surpasses it. This strategy works well for the shift of long stable patterns but can not quickly adapt to the interleaved patterns in graph applications. GraFetch achieves 74.97% of ideal performance while SBP only achieves 64.28% performance of the ideal prefetcher.

VIII. DISCUSSION

This paper explored DSML models to improve prefetching performance. While these models are not optimized for hardware implementation, we analyze the computational complexity of the predictors and discuss techniques for reducing model complexity that we will further explore.

A. Analysis of Computational Complexity

The DAMMA-based predictors in GraFetch dominate its computational complexity, while the detectors, using non-neural network methods, are lightweight. In Table X, we analyze the complexity of neural network predictors in GraFetch and state-of-the-art ML-based prefetchers using three metrics: total number of model parameters, model inference FLOPs, and critical path. Compared to LSTM-based methods Delta-LSTM and Voyager, GraFetch features a shorter critical path due to the high parallelism in attention. Compared with the attention-based method TransFetch, GraFetch employs phase-specific page and delta predictors. leading to a higher parameter count. As shown in Table X, GraFetch uses a total of 903.1 K parameters, comprising four separate models: two pairs of page and delta predictors corresponding to two distinct graph processing phases. This increase in model complexity, however, translates into a notable performance gain, with IPC improvement from 5.23% to 7.22%.

Using a fully parallel implementation of the DAMMA models, we can estimate the latency using the critical path based on Fig. 12 as follows:

$$T = \underbrace{T_{emb} + T_{att} + T_{fusion} + T_{trans}}_{\text{DAMMA}} + \underbrace{T_{hash}}_{\text{Input}} + \underbrace{T_{head} + T_{av}}_{\text{Output}} \tag{13}$$

where T_{emb} is for the embedding layer composed of a matrix multiplication (takes T_{mm}) and an activation function (takes T_{av}), T_{att} is for the self-attention layer that takes $4T_{mm}$ and $3T_{av}$, T_{fusion} is for the multi-modality attention fusion layer that takes $T_{att} + T_{mm} + 4T_{av}$, T_{trans} is for the Transformer layer with the same critical path with the fusion layer, T_{hash} is for the input processes (hashing, segmentation, and tokenization) that can be implemented in look-up tables with latency 1, and T_{head} is for the output layer that takes T_{mm}. Assuming full parallelism, $T_{mm} = 1 + \log_2 D$ for dimension D and $T_{av} = 1$ for activation functions using a look-up table, the overall latency is estimated as $T \approx 123$ processor cycles.

B. Techniques for Reducing Complexity

DSML-based predictors can be further optimized for more efficient hardware implementation by reducing the model storage cost and inference latency.

Reducing Storage Cost: As illustrated in Table X, GraFetch utilizes 903.1 K parameters. The storage cost can be reduced to approximately 0.86 MB under 8-bit quantization [73]. Although this storage requirement is higher than that of rule-based models, it remains well within acceptable limits for existing last-level cache controllers (several megabytes) [77]. Using knowledge distillation [72], we can train smaller DAMMA-based models as students using the current models as teachers while maintaining the performance. Tensor Decomposition [78] approximates the model weight matrices with lower-rank matrices to reduce the number of parameters. Emerging bio-inspired models such as

spiking neural networks [79] can be integrated into our HHP framework with smaller overheads.

Reducing Inference Latency: Recent work has explored approximating matrix multiplication and activation functions using tables [80], [81]. By implementing layer-wise table lookup, the model inference latency can be reduced to approximately 50 cycles. Distance prefetching [24] offers an alternative method to offset the latency by skipping the inference slot and predicting the memory accesses at a distance.

C. Online Model Update

Following existing ML-based prefetchers such as TransFetch [24] and Voyager [19]. This approach decouples the training cost from the deployment platform, ensuring portability and enabling optimizations tailored to the specific deployment environment. In real-world scenarios, the ML models require retraining to adapt to the dynamic allocation of processor and memory resources. Following steps can be used for model deployment and online update. We begin by training an initial pre-trained model offline using memory access data collected from a single iteration of the graph application. Offline training ensures that the model undergoes extensive optimization leveraging powerful hardware platforms such as GPUs. Then the pre-trained initial model is deployed on the hardware (e.g., on-chip memory on an embedded FPGA [29]) for online memory access prediction and data prefetching. During execution, we continuously collect memory access data and store the data in DRAM. For online model training, we load the training data from DRAM to high-bandwidth memory (HBM) and train a shadow model (stored in HBM) through hardware (e.g., compute units on FPGA [82]) for one epoch to adapt to evolving memory access patterns. After training, the updated models with 903.1 K parameters (as shown in Table X) are loaded into on-chip memory (e.g., BRAM) to enable fast inference.

D. Potential Hardware Deployment

Our prefetcher is not constrained to a specific existing architecture. Potential deployment involves integrating the existing memory system with emerging technologies, such as an embedded FPGA (eFPGA) [29] tightly integrated with a memory controller. An embedded FPGA (eFPGA) is a reconfigurable logic block integrated within a larger system-on-chip (SoC) or application-specific integrated circuit (ASIC). An example of hardware deployment using eFPGA is shown in Fig. 22. An eFPGA allows for reconfigurable hardware, making it adaptable to different prefetching models. The ML-based prefetching models are first compressed using techniques in Section VIII-A, resulting in simplified, quantized models or table-based approximations. The computations in the ML models are implemented using logic blocks within the eFPGA, including Multipliers, Adders, and Look-Up Tables. The parameters of the ML model are stored in on-chip memory, such as Block RAM (BRAM), providing fast, low-latency access during inference. There are also works exploring efficient ML model training on FPGA [83], [84], facilitating our model's online update. While the current implementation demonstrates the capability of our framework,

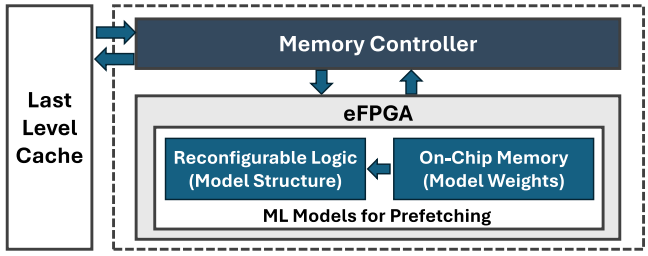


Fig. 22. Example hardware deployment of ML models for data prefetching in a Last-Level Cache (LLC) memory controller with eFPGA.

we acknowledge the tradeoff between the additional hardware resources required and the achievable performance. However, the framework is designed to provide a scalable foundation for future optimizations, such as distillation [85], quantization [38], and tabularization [80]. We believe that with further refinement, the resource cost can be significantly reduced, making the framework a practical solution for real-world GNN applications.

IX. Conclusion

In this paper, we proposed GraFetch, a domain-specific hybrid prefetching system designed to accelerate graph applications. Using a novel Hierarchical Hybrid Prefetching framework integrated with Domain-Specific Machine Learning models, GraFetch achieves an average of 12.47% IPC improvement on graph analytics and 4.17% IPC improvement on GNNs, outperforming state-of-the-art rule-based and ML-based prefetchers. In the future, we plan to optimize the model hardware implementation to reduce storage costs and latency overheads. We also intend to optimize the task allocation among multiple prefetchers and develop an adaptive controller to achieve more efficient prefetching. Additionally, we will study prefetching for the scenario of applications executing simultaneously. We will also explore the the minimum models required for achieving similar performance benefits.

Acknowledgment

We are grateful to Neelesh Gupta for his insightful perspectives in the development of this article. Distribution Statement A: Approved for public release. Distribution is unlimited

References

[1] S. Even, *Graph Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2011.
[2] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "GPOP: A scalable cache-and memory-efficient framework for graph processing over parts," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, pp. 1–24, 2020.
[3] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 472–488.
[4] B. Zhang, H. Zeng, and V. Prasanna, "GraphAGILE: An FPGA-based overlay accelerator for low-latency GNN inference," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 9, pp. 2580–2597, Sep. 2023.
[5] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1616–1637, Sep. 2018.
[6] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv: 1903.02428*.

[7] M. Wang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2019, *arXiv:1909.01315*.

[8] P. Zhang, R. Kannan, and V. K. Prasanna, "Phases, modalities, spatial and temporal locality: Domain specific ML prefetcher for accelerating graph analytics," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 1–15.

[9] J. Wu, J. Sun, H. Sun, and G. Sun, "Performance analysis of graph neural network frameworks," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2021, pp. 118–127.

[10] H. H. Liu, *Software Performance and Scalability: A Quantitative Approach*. Hoboken, NJ, USA: Wiley, 2011.

[11] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, 2000.

[12] P. Michaud, "Best-offset hardware prefetching," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 469–480.

[13] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–12.

[14] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2015, pp. 141–152.

[15] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 247–259.

[16] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 131–142.

[17] S. P. Vander Wiel and D. J. Lilja, "When caches aren't enough: Data prefetching techniques," *Computer*, vol. 30, no. 7, pp. 23–30, 1997.

[18] P. Zhang, A. Srivastava, T.-Y. Wang, C. A. De Rose, R. Kannan, and V. K. Prasanna, "C-MemMAP: Clustering-driven compact, adaptable, and generalizable meta-LSTM models for memory access prediction," *Int. J. Data Sci. Analytics*, vol. 13, pp. 3–16, 2022.

[19] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proc. 26th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2021, pp. 861–873.

[20] P. Zhang, R. Kannan, A. Srivastava, A. V. Nori, and V. K. Prasanna, "Re-Semble: Reinforced ensemble framework for data prefetching," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 282–292.

[21] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting memory accesses: The road to compact ML-driven prefetcher," in *Proc. Int. Symp. Memory Syst.*, 2019, pp. 461–470.

[22] M. Hashemi et al., "Learning memory access patterns," 2018, *arXiv: 1803.02329*.

[23] A. Vaswani et al., "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[24] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation for attention-based variable-degree prefetching," in *Proc. 19th ACM Int. Conf. Comput. Front.*, 2022, pp. 103–112.

[25] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends Cogn. Sci.*, vol. 3, no. 4, pp. 128–135, 1999.

[26] P. Branco, L. Torgo, and R. P. Ribeiro, "A survey of predictive modeling on imbalanced domains," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 1–50, 2016.

[27] L. Peled, U. Weiser, and Y. Etsion, "A neural network memory prefetcher using semantic locality," 2018, *arXiv: 1804.00478*.

[28] T. Baltrušaitis, C. Ahuja, and L.-P. Morency, "Multimodal machine learning: A survey and taxonomy," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 2, pp. 423–443, Feb. 2019.

[29] J. Gonski et al., "Embedded FPGA developments in 130nm and 28nm CMOS for machine learning in particle detector readout," 2024, *arXiv:2404.17701*.

[30] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 1–13.

[31] S.-W. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–10.

[32] S. Rahman, M. Burtscher, Z. Zong, and A. Qasem, "Maximizing hardware prefetch effectiveness with machine learning," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun.*, 2015, pp. 383–389.

[33] P. Zhang, A. Srivastava, B. Brooks, R. Kannan, and V. K. Prasanna, "RAOP: Recurrent neural network augmented offset prefetcher," in *Proc. Int. Symp. Memory Syst.*, 2020, pp. 352–362.

[34] G. Gerogiannis and J. Torrellas, "Micro-armed bandit: Lightweight & reusable reinforcement learning for microarchitecture decision-making," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2023, pp. 698–713.

[35] L. Peled, U. Weiser, and Y. Etsion, "A neural network prefetcher for arbitrary memory access patterns," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 1–27, 2019.

[36] Y. Zeng and X. Guo, "Long short term memory based hardware prefetcher: A case study," in *Proc. Int. Symp. Memory Syst.*, 2017, pp. 305–311.

[37] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "DeepCache: A deep learning based framework for content caching," in *Proc. 2018 Workshop Netw. Meets AI ML*, 2018, pp. 48–53.

[38] Q. Duong, A. Jain, and C. Lin, "A new formulation of neural data prefetching," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit.*, 2024, pp. 1173–1187.

[39] P. Zhang, N. Gupta, R. Kannan, and V. K. Prasanna, "Attention, distillation, and tabularization: Towards practical neural network-based prefetching," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2024, pp. 876–888.

[40] B. J. Oakes, M. Famelis, and H. Sahraoui, "Building domain-specific machine learning workflows: A conceptual framework for the state of the practice," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, pp. 1–50, 2024.

[41] R. J. Murdock, S. K. Kauwe, A. Y.-T. Wang, and T. D. Sparks, "Is domain knowledge necessary for machine learning materials properties?," *Integrating Mater. Manuf. Innov.*, vol. 9, pp. 221–227, 2020.

[42] K. Kashinath et al., "Physics-informed machine learning: Case studies for weather and climate modelling," *Philos. Trans. Roy. Soc. A*, vol. 379, no. 2194, 2021, Art. no. 20200093.

[43] J.-L. Wu, H. Xiao, and E. Paterson, "Physics-informed machine learning approach for augmenting turbulence models: A comprehensive framework," *Phys. Rev. Fluids*, vol. 3, no. 7, 2018, Art. no. 074602.

[44] S. Karimpouli and P. Tahmasebi, "Physics informed machine learning: Seismic wave equation," *Geosci. Front.*, vol. 11, no. 6, pp. 1993–2001, 2020.

[45] A. Karpatne et al., "Theory-guided data science: A new paradigm for scientific discovery from data," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 10, pp. 2318–2331, Oct. 2017.

[46] L. Von Rueden et al., "Informed machine learning–A taxonomy and survey of integrating knowledge into learning systems," 2019, *arXiv: 1903.12394*.

[47] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Rev. Phys.*, vol. 3, no. 6, pp. 422–440, 2021.

[48] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 178–190.

[49] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proc. Int. Conf. Supercomputing*, 2016, pp. 1–11.

[50] N. Talati et al., "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2021, pp. 654–667.

[51] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled vector runahead," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2023, pp. 17–31.

[52] J. Roelandts, A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Scalar vector runahead," in *Proc. 57th IEEE/ACM Int. Symp. Microarchitecture*, 2024, pp. 1367–1381.

[53] A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A hardware prefetcher for graph analytics," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, pp. 1–25, 2021.

[54] G. Fu, T. Xia, Z. Luo, R. Chen, W. Zhao, and P. Ren, "Differential-matching prefetcher for indirect memory access," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2024, pp. 439–453.

[55] Y.-C. Lin, B. Zhang, and V. Prasanna, "HitGNN: High-throughput GNN training framework on CPU+ multi-FPGA heterogeneous platform," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 5, pp. 707–719, May 2024.

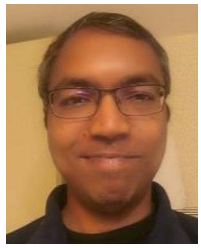
[56] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[57] P. Velickovic et al., "Graph attention networks," 2017, *arXiv:1710.10903*.

[58] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[59] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 5171–5181.
[60] P. Zhang, R. Kannan, X. Tong, A. V. Nori, and V. K. Prasanna, "SHARP: Software hint-assisted memory access prediction for graph analytics," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2022, pp. 1–8.
[61] A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown, "An introduction to decision tree modeling," *J. Chemometrics*, vol. 18, no. 6, pp. 275–285, 2004.
[62] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 1–49, 2017.
[63] M. E. Newman, D. J. Watts, and S. H. Strogatz, "Random graph models of social networks," in *Proc. Nat. Acad. Sci. USA*, vol. 99, no. suppl 1, pp. 2566–2572, 2002.
[64] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," in *Parallel Numerical Algorithms*. Berlin, Germany: Springer, 1997, pp. 323–368.
[65] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, pp. 5–32, 2001.
[66] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 12, pp. 2346–2363, Dec. 2019.
[67] C. Raab, M. Heusinger, and F.-M. Schleif, "Reactive soft prototype computing for concept drift streams," *Neurocomputing*, vol. 416, pp. 340–351, 2020.
[68] V. W. Berger and Y. Zhou, "Kolmogorov–Smirnov test: Overview," in *Wiley StatsRef: Statistics Reference Online*. Hoboken, NJ, USA: Wiley, 2014.
[69] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," 2018, *arXiv: 1810.00826*.
[70] N. Gober et al., "The championship simulator: Architectural simulation for education and competition," 2022, *arXiv:2210.14324*.
[71] D. M. Powers, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation," 2020, *arXiv: 2010.16061*.
[72] G. Hinton et al., "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*.
[73] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," 2018, *arXiv: 1802.05668*.
[74] V. Srinivasan, E. S. Davidson, and G. S. Tyson, "A prefetch taxonomy," *IEEE Trans. Comput.*, vol. 53, no. 2, pp. 126–140, Feb. 2004.
[75] S. H. Pugsley et al., "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 626–637.
[76] S. McFarling, "Combining branch predictors," Digit. Western Res. Lab., vol. 49, Tech. Rep. TN-36, 1993.
[77] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 1–35, 2016.
[78] S. Wijeratne, T.-Y. Wang, R. Kannan, and V. Prasanna, "Accelerating sparse MTTKRP for tensor decomposition on FPGA," in *Proc. 2023 ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2023, pp. 259–269.
[79] L. Jia, J. P. Mcmahon, S. Gudaparthi, S. Singh, and R. Balasubramonian, "PATHFINDER: Practical real-time learning for data prefetching," in *Proc. 29th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2024, pp. 785–800.
[80] P. Zhang, N. Gupta, R. Kannan, and V. K. Prasanna, "Attention, distillation, and tabularization: Towards practical neural network-based prefetching," 2023, *arXiv:2401.06362*.
[81] N. Gupta, N. Kannan, P. Zhang, and V. Prasanna, "TabConv: Low-computation CNN inference via table lookups," 2024, *arXiv:2404.05872*.
[82] S. K. Venkataramanaiah et al., "FPGA-based low-batch training accelerator for modern CNNs featuring high bandwidth memory," in *Proc. 39th Int. Conf. Comput. Aided Des.*, 2020, pp. 1–8.
[83] Y.-C. Lin, Z. Xu, and V. Prasanna, "xBS-GNN: Accelerating billion-scale GNNTraining on FPGA," in *Proc. Workshops Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2024, pp. 659–666.
[84] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2020, pp. 255–265.
[85] N. Gupta, P. Zhang, R. Kannan, and V. Prasanna, "PaCKD: Pattern-clustered knowledge distillation for compressing memory access prediction models," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2023, pp. 1–7.

Pengmiao Zhang received the BS degree in electrical engineering and automation from Northeastern University (China), in 2013, the MEng degree in electrical engineering from the Harbin Institute of Technology, in 2015, and the PhD degree in computer engineering from the University of Southern California, in 2024. He is currently working as a research scientist with Meta. His research interests include machine learning for computer systems, memory system optimizations, efficient machine learning, and recommendation systems.



Rajgopal Kannan received the BTech degree in computer science and engineering from IIT Bombay, in 1991 and the PhD degree in computer science from the University of Denver, in 1996. He is currently the Branch Chief and External Competency Co-Lead of Military Information Sciences at the DEVCOM ARL Army Research Office (ARO). He was formerly a professor with the Department of Computer Science, Louisiana State University (2000-2015). His academic research was funded by DARPA, NSF and DOE and he has published more than 150 research papers in international journals and conferences with two patents awarded in the area of network optimization. His research interests are at the intersection of graph analytics, machine learning and edge computing - enabling application acceleration at the edge on low power devices and also in AI for decision support.

Viktor K. Prasanna (Fellow, IEEE) received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is Charles Lee Powell chair in engineering with the Ming Hsieh Department of Electrical and Computer Engineering and professor of computer science with the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He serves as the director of the Center for Energy Informatics, USC. He served as the editor-in-chief of the *IEEE Transactions on Computers* during 2003–06. Currently, he is the editor-in-chief of the *Journal of Parallel and Distributed Computing*. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University, the 2019 Distinguished Alumnus Award from the Indian Institute of Science (IISc) and the 2016 Distinguished Alumnus Award from the University Visvesvaraya College of Engineering (UVCE), Bangalore University. He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. He is a fellow of the ACM, and the American Association for Advancement of Science (AAAS). He is an elected member of Academia Europaea.