A New Formulation of Neural Data Prefetching

Quang Duong
The University of Texas at Austin
qduong@cs.utexas.edu

Akanksha Jain
Google
akanksha@cs.utexas.com

Calvin Lin
The University of Texas at Austin
lin@cs.utexas.edu

***Abstract*—Temporal data prefetchers have the potential to produce significant performance gains by prefetching irregular data streams. Recent work has introduced a neural model for temporal prefetching that outperforms practical table-based temporal prefetchers, but the large storage and latency costs, along with the inability to generalize to memory addresses outside of the training dataset, prevent such a neural network from seeing any practical use in hardware.**

In this paper, we reformulate the temporal prefetching prediction problem so that neural solutions to it are more amenable for hardware deployment. Our key insight is that while temporal prefetchers typically assume that each address can be followed by any possible successor, there are empirically only a few successors for each address. Utilizing this insight, we introduce a new abstraction of memory addresses, and we show how this abstraction enables the design of a much more efficient neural prefetcher.

Our new prefetcher, Twilight, improves upon the previous state-of-the-art neural prefetcher, Voyager, in multiple dimensions: It reduces latency by $988\times$, shrinks storage by $10.8\times$, achieves 4% more speedup on a mix of irregular SPEC 2006, SPEC 2017, and GAP benchmarks, and is capable of predicting new temporal correlations not present in the training data. Twilight outperforms idealized versions of the non-neural temporal prefetchers STMS by 12.2% and Domino by 8.5%. While Twilight is still not practical, T-LITE, a slimmed-down version of Twilight that can prefetch across different program runs, further reduces latency and storage ($1421\times$ faster and $142\times$ smaller than Voyager), matches Voyager's performance and outperforms the practical non-neural Triage prefetcher by 5.9%.

I. Introduction

Data prefetchers are vital mechanisms for hiding the long latencies of memory accesses. While most modern hardware prefetchers identify strides or spatial footprints to target regular or spatial access patterns, this paper focuses on temporal prefetching, a type of irregular data prefetching that identifies pairs of addresses that are temporally correlated. For instance, if address X is often followed by address Y, then X and Y are correlated, and a load of X can serve as a trigger to prefetch Y. Since these correlations can be found between any two addresses X and Y, temporal prefetchers can eliminate cache misses from any arbitrary repeated memory access stream.

Recent work by Shi, et al. [41] shows that ML-based temporal prefetchers like Voyager provide significant headroom over idealized versions of table-based temporal prefetchers, achieving an extra 13% speedup.[1] Unfortunately, several limitations prevent Voyager from being realized in practice.

[1]More compellingly, Shi, et al. show that on proprietary Google Search and Ads workloads, Voyager sees accuracy and coverage far beyond what non-neural prefetchers can obtain (73.9% vs 51.1%). Without access to these proprietary workloads, we report results from publicly available workloads.

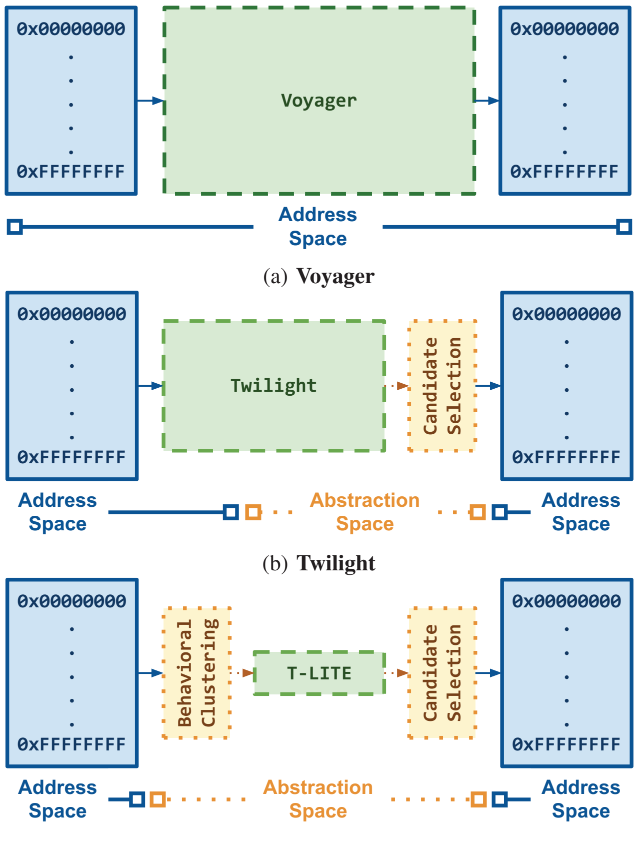


Fig. 1: Overview of model inputs and outputs. Instead of operating on data addresses directly like Voyager, Twilight's and T-LITE's neural models utilize layers of indirection to create an abstraction space that improves their practicality.

Excessive Model Size and Latency: Figure 1a shows that Voyager takes as input and produces as output any address in the address space. As a result, the neural model's size and prediction latency grow with the program's memory footprint. Across an irregular subset of the GAP, SPEC 2006, SPEC 2017 and Google server workloads, Voyager requires on average 81M FLOPs to make a prediction and 113.5 MB of storage.

Inability to Adapt or Generalize: Voyager was designed as a limit study, and there is no clear path for it to be trained in a realistic setting. On one hand, Voyager cannot be trained online while the program runs because it requires multiple passes over the training data and billions of FLOPS to learn each correlation. On the other hand, it cannot be trained offline because it can only learn temporal correlations for data addresses that it sees during training, which is problematic because the virtual to physical address mapping changes on every program execution, rendering Voyager's offline learned address correlations useless.

Unfortunately, these two issues are so significant that they cannot be mitigated by standard machine learning techniques. For example, distillation [22] and quantization [18] would only shrink the model size by some small factor (up to $8\times$ for quantization). Thus, if we are ever to see practical deployment of a neural temporal prefetcher, we need to rethink the prediction problem so that it is more amenable to a neural solution. This paper does precisely that.

We observe that the root cause of these issues is the very definition of temporal prefetching, which correlates addresses with other addresses. This formulation results in huge storage and computational costs, and it limits generalization since Voyager cannot predict addresses outside of the training data. Since addresses are the problem, we reformulate the temporal prefetching prediction problem to avoid their use.

Our key observation, which we refer to as the *Sparse Connectivity Hypothesis*, is that in practice each address is only ever succeeded by a handful of other addresses. Armed with this insight, we reformulate the problem: Instead of predicting an address, the neural model predicts an ordinal i, which corresponds to prefetching the i^{th} most frequent prefetch candidate. For example, suppose A is followed by B 40% of the time and is followed by C 60% of the time. If the model predicts $i = 1$, then the most frequent candidate, C, is prefetched; and if the model predicts $i = 2$, then the second most frequent candidate, B, is prefetched. We refer to this new formulation as *frequency-based candidate selection*, and we use it to create an abstraction layer that insulates our new neural prefetcher, which we refer to as Twilight,[2] from the output address space (see Figure 1b).

Our new formulation plays a central role in the design of a slimmed down version of Twilight, referred to as Twilight-LITE or simply T-LITE, which moves us closer to a practical neural temporal prefetcher. In particular, candidate selection enables us to create an abstraction layer between T-LITE and the input address space by grouping together addresses with similar prefetching behavior (see Figure 1c). We find these groups by using offline clustering to divide the address space into a fixed set of *behavioral clusters.* Together, our two new abstraction layers insulate T-LITE from interacting directly with memory addresses, so T-LITE's size and latency do not grow with the program's memory footprint, making T-LITE small and fast enough to be in the realm of feasible hardware deployment.

These two abstraction layers provide another advantage: They allow T-LITE's deployment to be staged. The neural model is trained offline, while the mappings between the address space and the new abstraction layers are learned online. That is, at runtime, T-LITE tracks the set of addresses that comprise the set of prefetch candidates and dynamically assigns unseen addresses to behavioral clusters. As a result, T-LITE better adapts across program phases and can prefetch for subsequent runs across different program inputs.

[2]The name Twilight reflects the fact that it sits at the edge of the offline and online divide just as twilight separates night and day.

This paper makes the following contributions:

- We reformulate the notion of temporal prefetching by introducing two novel levels of indirection, namely, *frequency-based candidate selection* and *behavioral clustering*, which abstract the neural model's outputs and inputs, respectively.
- We present Twilight, a neural temporal prefetcher that utilizes *frequency-based candidate selection* to provide the following benefits over Voyager:
  - $988\times$ faster inference (82K FLOPs).
  - $10.8\times$ smaller neural model (10.5 MB).
  - 4% higher IPC improvement over no prefetcher.
  - Better generalization because its online component dynamically tracks metadata for prediction. Whereas Voyager loses 17% performance on unseen program execution, Twilight loses just 6.6%.
- We present T-LITE, a slimmed down version of Twilight that utilizes *behavioral clustering* to trade off performance for significantly increased practicality.
  - T-LITE matches Voyager's performance while being $1421\times$ faster (57K IOPs) and $142\times$ smaller (0.8 MB).
  - T-LITE sees 5.9% more speedup than the practical table-based Triage prefetcher, which stores up to 1 MB of metadata in the cache. By contrast, T-LITE requires just 64 KB of metadata by baking in some of the prefetching knowledge into the neural model.
  - T-LITE's staged deployment transfers the knowledge learned in offline training across program phases and program inputs. On GAP benchmarks that are run on inputs from unseen domains, T-LITE outperforms Triage by 16.6% and achieves 94% of the performance of a T-LITE model fine-tuned on the unseen inputs.

The remainder of this paper is organized as follows. We place our work in the context of prior work in Section II. We then describe our solution in Section III and our experimental methodology in Section IV before presenting our evaluation of Twilight in Section V and T-LITE in Section VI. We discuss future work in Section VII and conclude in Section VIII.

II. Related Work

Data prefetching [8], [9], [12], [25], [29], [31], [32], [34], [38], [39], [43], [50], [52], [53] has been studied for decades, but the use of machine learning is fairly recent, with almost all work in this area [13], [21], [36], [45], [46], [55]–[60] focusing on delta prefetching, which is suitable for programs with good spatial locality. Peled, et al [36] explore neural models for irregular data prefetching, but their solution has poor predictability for irregular programs because it phrases the problem as delta prefetching. Voyager [41] is the only prior solution that supports temporal prefetching.

We now discuss related work in more detail by first discussing ML-based spatial prefetching and then discussing both non-neural and neural solutions for temporal prefetching. We end by briefly discussing the use of profiling in data prefetching and other applications of ML for hardware prediction.

A. *ML-Based Spatial Prefetching*

Most work in ML prefetching focuses on using neural networks [13], [21], [36], [45], [46], [55]–[60] or reinforcement learning (RL) [12], [35] to improve the prediction accuracy and coverage of delta prefetchers. To make costly neural networks more amenable for evaluation and deployment, neural stride prefetchers have used (1) clustering to share model weights across address regions [21] and across different applications [46], [60], (2) compression by representing deltas in binary format [36], [45], [46], [57], [60], (3) spatial bitmask prediction [6], [55], [56], [58], [59], and (4) graph specialization [55], [56]. Prefetchers based on RL [12], [35] have significantly lower cost in exchange for lower predictive power. However because these approaches can train online, they recover this performance loss by adapting better to changes in program behavior.

Our work is orthogonal to this prior work because it focuses on temporal prefetching, which has a much larger problem space than spatial prefetching; our new abstraction shrinks the problem space, which in turn shrinks the neural model to be closer in size to those for neural spatial prefetching.

B. *Temporal Prefetching*

The notion of temporal prefetching was introduced by Joseph and Grunwald, who use a table to record multiple successors for a given memory address [28]. Subsequent work has followed two broad directions, with one focused on performance and the other on practicality.

The first direction improves the performance of temporal prefetchers by devising more sophisticated prediction mechanisms, such as spatio-temporal correlation [44], PC-localization [25], and longer histories [8], [41]. The Voyager neural prefetcher [41] uses a long history of data addresses to significantly outperform prior art.

The second direction addresses the large metadata requirements of temporal prefetchers. Early solutions store the correlation table off-chip and optimize the memory bandwidth requirements and prefetch look-ahead distance for off-chip table access [15], [42]. The Global History Buffer [33] amortizes the cost of accessing off-chip metadata for long streams [50]. The ISB prefetcher [25] reformulates temporal prefetching so that its metadata can be cached on-chip, and the MISB prefetcher [53] shows that this off-chip metadata can be easily prefetched to hide its off-chip access latency. Finally, the Triage prefetcher [52] stores the most relevant metadata on-chip by repurposing a portion of the last-level cache.

Twilight significantly reduces the high latency and storage costs of prior neural temporal prefetchers. Furthermore, Twilight is more generalizable because it incorporates an online component that enables it to adapt to unseen data addresses.

C. *Neural Temporal Prefetching*

Shi et al. [41] formulate temporal prefetching as a classification task and show that temporal prefetching suffers from a *class explosion problem* where the sheer number of classes—2^{52} possible cache lines in a 64-bit address space—would make any neural solution untenable. Because the number of unique pages encountered in their program traces is $1-2$ orders of magnitude smaller than the number of unique cache lines, Shi, et al. split cache line addresses into page and offset pairs, enabling their Voyager model to be feasibly trained.

We observe that while the page-offset split enables offline training over a small region of 250M instructions, the underlying *class explosion problem* has not been solved. At best, this split reduces the number of classes by $64\times$, so the number of classes (2^{46}) remains impractically large.

For example, there are SPEC 2006 traces that Voyager was not evaluated on with as many as 255K unique pages within a 250M instruction span, which is approximately $3\times$ larger than the biggest footprint Voyager was evaluated on. Since Voyager's size and latency grow with the number of unique pages, the neural model for this trace takes a week to make a single pass over the data and is unable to exceed 1% validation accuracy even after two months of training.

D. *Profile-Driven Dynamic Prediction*

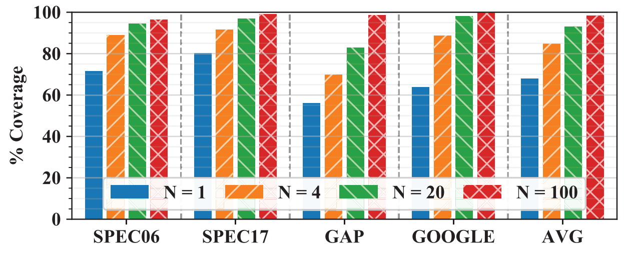
T-LITE's staged deployment might seem to resemble previous profile-driven prediction mechanisms, but profile-driven prefetchers [26] use profiling to insert software prefetches into the code, which adds instruction overhead and is difficult to time correctly across different program runs. T-LITE's use of profiling is different—it uses offline training to augment hardware prefetching. In particular, T-LITE has two stages: (1) an offline stage that trains the neural model on profiled memory traces, and (2) an online stage (that executes on the hardware) in which T-LITE dynamically collects runtime information that informs the offline trained model and enables it to be reusable across program runs. A similar strategy has been used in neural branch prediction [54].

E. *ML for Other Hardware Prediction Tasks*

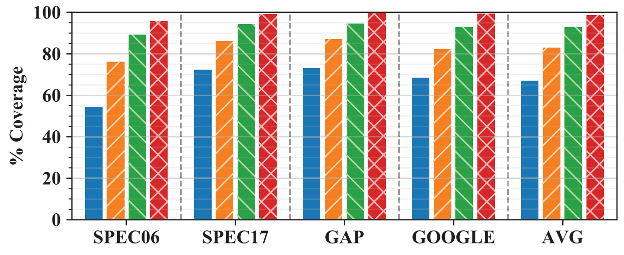
Machine learning has been used for other hardware prediction problems, such as cache replacement [30], [37], [40], [48], branch prediction [47], [54], and level prediction [11], but such work is largely orthogonal to ours.

III. Our Solution

This section presents our reformulation of the temporal prefetching problem. We explain our two new abstraction layers, we provide an overview of Twilight's neural architecture, and we describe how it generates a prefetch address.



(a) **Cache Line-to-Cache Line**



(b) **Page-to-Page**

Fig. 2: The top graph shows the percentage of cache lines that could be prefetched when limited to the top-N most frequent cache lines after a given trigger cache line. The bottom graph shows similar results for pages. For $N = 20$, 93.2% of cache lines / 92.7% of pages are contained within this set of the top-N successors.

A. *Problem Formulation*

Temporal prefetching, also known as address correlation, typically predicts the address that temporally follows a history of addresses. Under this formulation, the size and computational cost of a neural prefetcher grow with the size of the program's memory footprint: For each unique address, the neural model needs an embedding[3] to represent that address as input and an output neuron to predict that address as a prefetch candidate.

1) The Sparse Connectivity Hypothesis: We observe that a given cache line is typically followed by only a small set of other cache lines. Figure 2a empirically supports this claim: 68% of the cache lines are followed by their most likely successor, where a *successor* is defined as the address that immediately follows a given history of accesses. 93.2% of cache lines are followed by one of their top 20 successors, and 98.5% of accesses are followed by one of their top 100.

2) Frequency-Based Candidate Selection: To insulate the neural model from the huge output space of addresses, our new formulation introduces the notion of *frequency-based candidate selection*: Based on the Sparse Connectivity Hypothesis, we constrain the prefetcher to select from among the top-N most likely successors. To account for the case where the ground truth address is not among these N candidates, we add an $(N+1)^{th}$ option so that the prefetcher can learn when not to prefetch.

With *frequency-based candidate selection*, the output of our neural model is an ordinal value i—where i corresponds to the i^{th} most frequent successor, or $N+1$ for no prefetching—rather than the address itself. Thus, the output space of our neural temporal prefetcher has constant size—just $N+1$ output neurons—instead of growing with the memory footprint. In Section V-D3, we show that the neural network is able to learn a non-trivial distribution of successors, and we show that the learned distribution is fairly close to the true distribution.

3) Page Granularity: Shi, et al. [41] find that the number of unique cache lines grows far quicker than the number of unique pages and would require an infeasible amount of storage and compute, so decomposing data addresses into their respective pages and offsets enables a significantly more tractable neural model. In the context of *frequency-based candidate selection*, we observe in Figure 2b that our *Sparse Connectivity Hypothesis* extends to pages as well: A given page will likewise be succeeded by a small set of pages. Thus, similar to Voyager, Twilight decomposes the address prediction problem into page prediction and offset prediction. Because there are only a small fixed number of cache line offsets (64), we only apply our new problem formulation to Twilight's page prediction and continue to predict offsets directly.

B. *Behavioral Clustering*

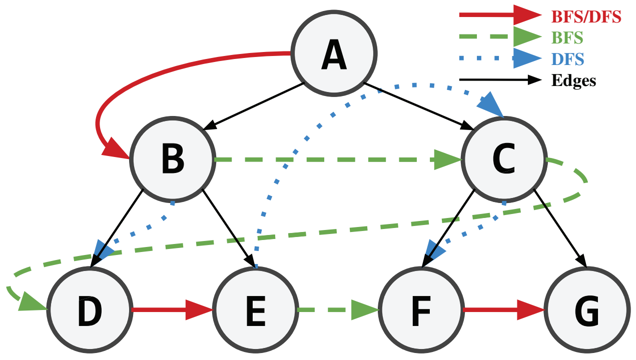
To insulate the neural model from the huge input space of addresses, our new formulation introduces the notion of *behavioral clustering*, which groups together pages that have similar prefetching behavior.

1) Graph Traversal Example: Under traditional address correlation, A being followed by X and B being followed by Y are unrelated correlations, but under candidate selection, A and B would now be similar if both were followed by their j^{th} candidate in one context and their k^{th} candidate in another. To understand why such groupings might be useful, consider the example shown in Figure 3, in which a graph program performs BFS 70% of the time and DFS 30% of the time, which leads to two clusters of nodes:

1) A, D, F where the next accessed node is the same for both BFS and DFS
2) B, C, E where the next accessed node differs between BFS and DFS

For cluster (1), our prefetcher would learn to always prefetch the $i = 1$ node for all contexts. For cluster (2), the prefetcher would learn to prefetch the $i = 1$ node for PCs that are performing BFS and the $i = 2$ node for PCs that are performing DFS. Armed with this insight, we can cluster pages based on their prefetching behavior and then replace the input history of pages with the corresponding history of clusters without a significant reduction in prediction accuracy.

[3]An embedding is a learned representation commonly used to represent categorical variables (such as words, addresses or in our case, pages or behavioral clusters) as a numerical vector such that similar categories would have similar embeddings.



(a) Example Graph

Node	Most Freq	2nd Most Freq	Cluster
A	B (100%)	-	1
B	C (70%)	D (30%)	2
C	D (70%)	F (30%)	2
D	E (100%)	-	1
E	F (70%)	C (30%)	2
F	G (100%)	-	1

(b) Top-2 most frequent succeeding nodes for each node

Fig. 3: Graph traversal example where a program executes **BFS** 70% of the time and **DFS** 30% of the time. Two clusters of nodes arise: (A, D, F) and (B, C, E), where the former prefetches the same address regardless of the traversal pattern and the latter prefetches differently depending on the context.

2) Empirical Validation: To empirically evaluate the idea that behavioral clusters are an effective substitute for pages, we examine whether pages with "similar" prefetching behavior produce similar predictions. Page embeddings implicitly capture prefetching behavior, so we examine Twilight's trained page embeddings to see whether similar embedding vectors produce similar offset predictions. For a given sample trace, we track the offset predictions for each page in a 64×64 matrix of the page's *offset transitions*. Each entry (i, j) in a page's offset transition matrix corresponds to the number of times that a cache line in that page with offset i is followed by another cache line with offset j.

Figure 4 shows a t-SNE visualization [49] for omnetpp where each point represents one of the page embeddings; points that are close together have similar page embeddings. The shape and color of each point is determined by a k-means clustering of the *offset transitions* matrix of that point's page. The figure clearly shows that pages with similar embeddings are often within the same cluster of *offset transitions*, empirically confirming that different pages will have similar prediction behaviors under *frequency-based candidate selection*.

3) Clustering: We compute a set number of behavioral clusters, which yields a fixed-sized page embedding table and a fixed-sized neural model. Section VI-B1 shows empirically that this clustering produces minimal performance loss.

With *behavioral clustering*, the training procedure for

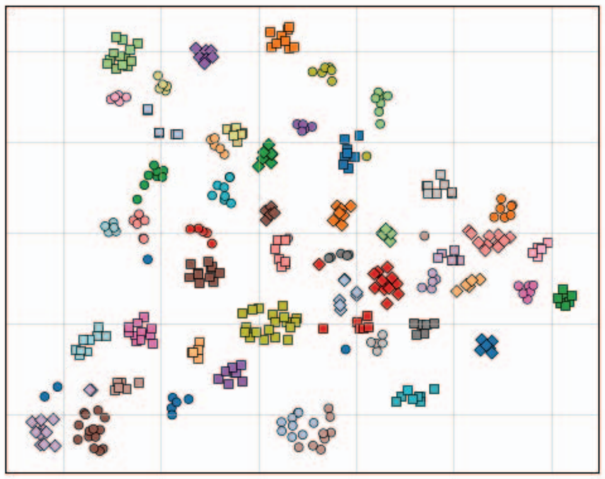


Fig. 4: A t-SNE [49] visualization of a subset of Twilight's page embeddings for a SPEC 2006 omnetpp trace. The color and the shape correspond to the clustering of the page's *offset transitions*, demonstrating that pages that are similar in the embedding space produce similar offset predictions.

T-LITE occurs in two passes. First as in Voyager, the entire neural model with per-page embeddings is trained offline. Then, we cluster the learned page embeddings using k-means and force pages in the same cluster to share an embedding, which we initialize to the centroid. Finally, we fine-tune all of the model weights including the embeddings by retraining on the same training data under these constraints to recalibrate the model to using *behavioral clusters* instead of pages.

C. Dynamic Metadata Collection

Our new problem formulation requires maintaining, for every input history, the distribution of the N most frequent successor pages (see Table 3b for an example distribution). Since these distributions change throughout a program's execution, we dynamically track their metadata similar to how non-neural temporal prefetchers [52], [53] track address correlations. Namely, each metadata entry corresponds to an input history and stores the frequencies for each of its successors. However, naively tracking metadata for every history has exponential cost: For a history of H prior pages, there could be $O(P^H)$ distributions where P is the number of unique pages. To avoid this exponential increase in metadata, we introduce the notion of *decoupled positional frequency* or DPF.

We define DPF as the frequency distribution of successors that occur several accesses ahead. More formally, let $f(h, n)$ denote the distribution of successors that are n accesses ahead of the history h. Rather than tracking a per-history joint distribution, $f((X, Y, Z), 1)$, DPF decouples the history and tracks $f(X, 3)$, $f(Y, 2)$ and $f(Z, 1)$ separately which scales better with $O(P \times H)$ distributions. We can then approximate

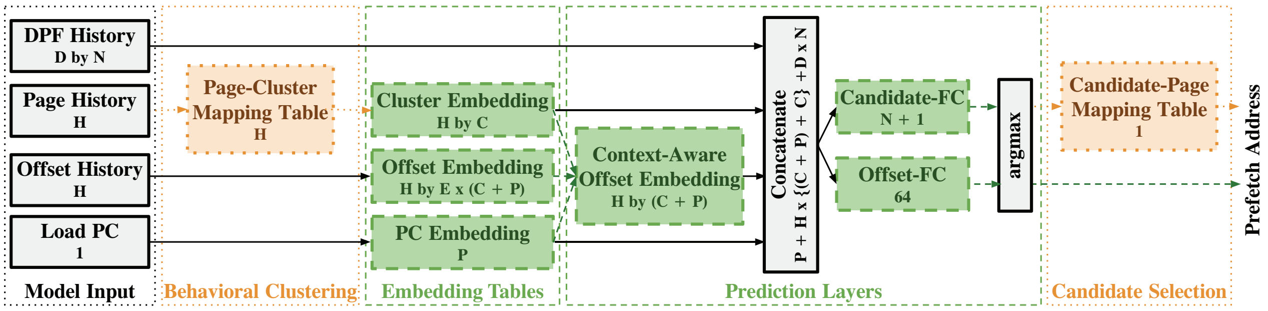


Fig. 5: T-LITE's Neural Architecture. Neural components are **dashed** and abstraction layers are **dotted**. The dimensionality of each component's outputs is labeled above. See Table I for the value of each hyperparameter (H, N, etc). Twilight has a similar organization but without the behavioral clustering, and it utilizes page embeddings instead of cluster embeddings.

Component	Hyperparameters
Neural Model	• PC Embedding Dim [P] = 64 • Page/Cluster Embedding Dim [C] = 25 • # of Offset Experts [E] = 100 • History Length [H] = 3 • Max # PC Embeddings = 4096 (no max)
Abstraction	• # Candidates [N] = 4 (20) • DPF History Length [D] = 1 (3) • # Behavioral Clusters = 4096
Training	• Learning Rate / Decay = 0.001 / 0.5 • Batch Size = 256 • # Epochs = 500 • # Early Stopping Epochs = 50 • Optimizer = Adam

TABLE I: T-LITE Hyperparameters. Twilight-specific hyperparameters are parenthesized.

the true joint distribution by multiplying the individual DPFs:

$$f((X,Y,Z),1) \approx f(X,3) \times f(Y,2) \times f(Z,1)$$

However, the weight of each access is not uniform in practice—the most recent access is typically the best predictor for prefetching. We feed in the raw DPF distributions to our neural models which learn the best weights to assign to each access. Section III-D shows an example of how DPF is used as a model input and how it determines the prefetch candidates.

D. Model Overview

Figure 5 shows T-LITE's (and Twilight's) overall architecture and Table I contains its hyperparameters. As per Section III-A, T-LITE predicts the prefetch offset directly and an ordinal i that maps to a prefetch candidate page. T-LITE decomposes the input address history into a page history and an offset history and maps these incoming pages to behavioral clusters as described in Section III-B. T-LITE also takes in the raw DPF distributions as input, as described in Section III-C.

We utilize a mixture-of-experts [24] approach similar to Voyager's *page-aware offset embeddings* to generate our *context-aware offset embeddings* that incorporate page and PC information into the offset embeddings. Each raw offset embedding is subdivided into multiple experts associated with the different contexts in which the offset appears. The context—the page and the PC—weights the information from each of the experts via attention [7] and combines them into the final *context-aware offset embedding*.

Conceptually, Voyager's *page-aware offset embedding* represents a given cache line because it combines the page embedding and the offset embedding. By contrast, since T-LITE's *context-aware offset embedding* additionally incorporates the load PC, it further contextualizes this cache line embedding to contain information about the access stream in which this particular access occurred.

T-LITE utilizes a single fully connected layer for candidate prediction (Candidate-FC) and another for offset prediction (Offset-FC) for a much more compact and parallelizable design than Voyager's pair of LSTMs that require significant serialized computation. Furthermore, Twilight's page/cluster embeddings (25D) are an order of magnitude smaller than Voyager's page embeddings (256D), and to constrain the size of the PC embedding table, we limit the number of load PC embeddings to the 4096 most occurring load PCs.

The *candidate-page mapping table* is stored implicitly in the DPF metadata for a given page. Furthermore, we augment each page's DPF metadata entry to include a 12-bit integer that maps that page to a *behavioral cluster* which obviates the need for a dedicated *page-cluster mapping table*.

E. Model Inference

We now show an example of a T-LITE prediction using the graph traversal example from Figure 3. Recall that $f(A,n)$ corresponds to the distribution of successors n accesses ahead of A. For a given input history, (B,D,E), the most recent access E determines the set of prefetch candidates: $\{F,C\}$. We construct the DPF input vectors utilizing only those candidates, and we then renormalize the resulting vectors:

$$f(B,3) = [F:0.0, C:1.0]$$
$$f(D,2) = [F:0.7, C:0.3]$$
$$f(E,1) = [F:0.7, C:0.3]$$

Since D is not one of the prefetch candidates, the DPF vector for $f(B,3)$ normalizes the frequency of the candidate

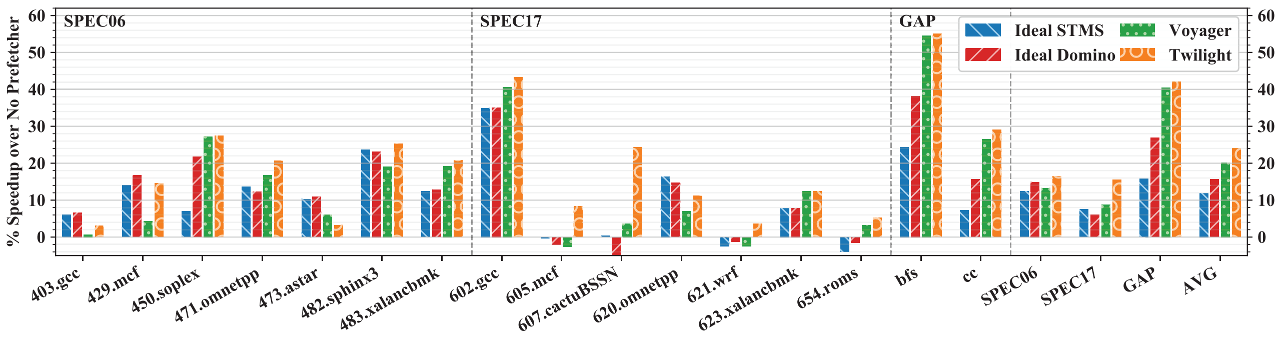


Fig. 6: Single Core Speedup comparing temporal prefetchers across SPEC 2006, SPEC 2017, and GAP benchmark suites

C to 1. Next, we feed the DPF vectors, cluster sequence, offset sequence and load PC to T-LITE. T-LITE computes the *context-aware offset embedding* and concatenates it with the above DPF vectors, the raw cluster embeddings, and the raw PC embedding. We flatten this vector and feed it into the two fully connected layers for prediction. For candidate prediction, a prediction of 1 maps to the most frequent page, F, 2 maps to the second most, C, and 3 maps to no prefetch. Since $f(B, 3)$ indicates that F has never occurred 3 accesses ahead of B, T-LITE would likely select C. We then combine the selected page with the predicted offset to get the prefetch address.

IV. Methodology

Core	Out-of-order, 4 GHz, 352 ROB entries 6-wide fetch, decode, and dispatch perceptron-based branch predictor [27]
L1I	32 KB, 8 ways, 4-cycle latency, 8 MSHRs
L1D	48 KB, 12 ways, 5-cycle latency, 16 MSHRs
L2	512 KB, 8 ways, 10-cycle latency, 32 MSHRs
LLC	2 MB / core, 16 ways, 20-cycle latency, 64 MSHRs
DRAM	3200 MT/S, 8 B channel width, 64K rows tCAS = tRP = tRCD = 12.5, 8 banks / rank 1/4/8-Core: 1/2/4 channels, 1/2/2 ranks / channel

TABLE II: Machine Configuration

Voyager [41] simulated online training with epochs of 50M instructions—the model trained on one epoch and evaluated on the next epoch. However, since neither Voyager nor Twilight can feasibly train online, we instead utilize an offline training scheme to more faithfully model staged deployment. In particular, we train the model on one region of a SimPoint [20] and evaluate on an unseen region of the same SimPoint.

All of our evaluated neural and non-neural prefetchers reside in the LLC, training on the LLC access stream and prefetching with degree 1. For fairness, the hardware prefetchers warm up their structures on the neural model's training region.

Simulator: We simulate the baseline and neural prefetchers using ChampSim [19], a trace-based simulator that models a 6-wide OOO processor with a 352-entry re-order buffer. ChampSim has a detailed memory system with a three-level cache hierarchy. Table II details the system configuration, which is similar to an Intel Ice Lake core [3].

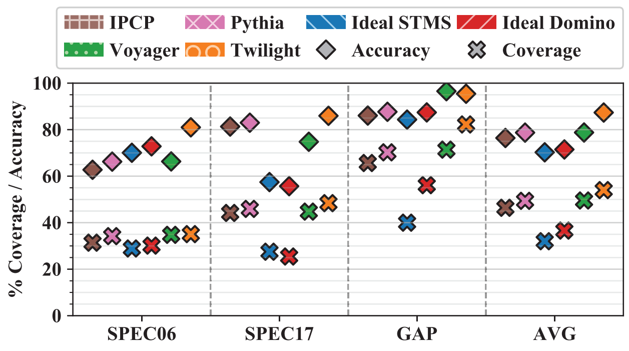
Benchmarks: We use as benchmarks the GAP benchmark suite [10] and an irregular memory-intensive subset from the SPEC 2006 CPU [1] and SPEC 2017 CPU [2] suites (we select workloads that have $>5\%$ IPC improvement with an idealized temporal prefetcher). Since the Google workloads that Voyager was evaluated on are proprietary, we instead report results for a subset of the recently released Google server workloads [5].

For SPEC, we use all SimPoints for each benchmark in our subset. For the GAP benchmarks, we use as inputs synthetic graphs with 2^{17} vertices for all results except in Section VI-E2 where we evaluate T-LITE's transferability across input graph domains: web crawls, road networks, and citation networks.

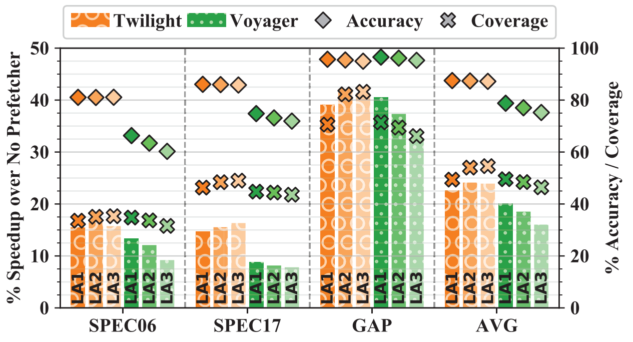
For both of SPEC and GAP, the neural models train on the first 200M instructions of the trace, validate on the next 25M instructions, and are then evaluated on the subsequent 175M instructions. Unless stated otherwise, all speedup results are from the 175M region. By evaluating about $4\times$ the number of instructions evaluated by Voyager [41], we are able to better gauge the model's ability to generalize to unseen data.

For the Google server workloads, we port to ChampSim the original traces, which are provided under the DynamoRIO simulation infrastructure [14]. The memory dependence information is unavailable, but we model them using statistics collected from CloudSuite [17], another server benchmark suite. For each benchmark, we select the longest thread, and warm up on the first 80% and evaluate on the last 20%. Because of the missing information, we only evaluate prefetcher accuracy and coverage for these workloads.

Baselines: In Section V, we compare Twilight against the state-of-the-art neural temporal prefetcher, Voyager [41], and two idealized non-neural temporal baselines, Domino [8] and STMS [50]. For these temporal prefetchers, we focus on evaluating predictive power, so we do not simulate prediction latency, storage cost, or off-chip metadata traffic. Furthermore, all temporal prefetchers are evaluated with lookahead 2, which provides better timeliness, except for Voyager which performs better at lookahead 1 (see Figure 7b). We also compare against two delta prefetchers: IPCP [34], the DPC-3 winner, and Pythia [12], an ML-based prefetcher.



(a) Prefetch Accuracy (⧫) and Coverage (**X**)



(b) Varying Prefetch Lookahead (LA)

Fig. 7: (a) Twilight has similar or better coverage (**X**) at greater accuracy (⧫) across all benchmark suites. (b) Twilight's superior accuracy enables larger lookahead (**LA**) prefetching which improves timeliness and coverage.

Whereas in Section V we evaluate temporal prefetchers idealistically, in Section VI we more faithfully model T-LITE's latency and storage to more fairly compare with Triage, a practical table-based temporal prefetcher with on-chip metadata.

V. Twilight Evaluation

We first present our single-core results. Figure 6 shows Twilight's performance compared with the baseline prefetchers across SPEC and GAP. We make several observations.

First, Twilight (24.1%) outperforms Voyager (20.1%) on all benchmark suites despite being orders of magnitude smaller and faster (see Figure 16a). Twilight's speedup on 473.astar is lower than Voyager because Twilight's set number of candidates, $N = 20$, cannot accommodate the wider successor distributions found in this benchmark. Voyager performs worse than Twilight on 403.gcc, mcf, 607.cactuBSSN, and 621.wrf because these benchmarks have many accesses in their evaluation region to pages not in the training data. As a result, Voyager cannot effectively prefetch for those benchmarks because its offline learned address correlations are useless, whereas Twilight's dynamically collected metadata allows it to adapt and prefetch these new pages.

Second, Twilight (24.1%) significantly outperforms the non-neural baselines, idealized STMS (11.9%) and idealized Domino (15.6%) in all benchmark suites. By contrast, Voyager performs worse than the non-neural baselines on SPEC due to its inability to generalize across longer evaluation periods (see Section V-D2). While our new abstractions dramatically improve generalization, Twilight underperforms the non-neural baselines on some of the benchmarks because there are some access patterns and load PCs not found in the training data. Training Twilight with more representative datasets or fine-tuning the neural model online with a low-cost solution like LoRA [23] could help bridge the remaining performance gap. Low-Rank Adaptation (LoRA) fine-tunes neural models by separating the fine-tuning weight updates from the model and then decomposing them into two low-rank matrices which enables cheap computation and requires minimal storage.

Figure 7a shows that Twilight achieves coverage similar to or better than other prefetchers while also having significantly higher accuracy across benchmark suites. In particular, Twilight has 8.6% higher accuracy than the next best prefetcher, Voyager. Twilight's higher accuracy is the direct result of its *frequency-based candidate selection,* which culls pages that are likely to be useless. By contrast, since Voyager can prefetch any page, it periodically issues prefetches to pages entirely unrelated to the current access stream.

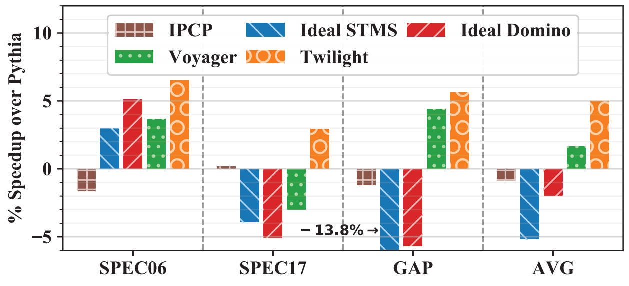
In Figure 7b, we observe that while Voyager's accuracy declines with increasing lookahead, Twilight's accuracy is constant, enabling it to achieve 1.5% more speedup due to better timeliness at higher lookahead. Normally, the further into the future a prefetcher attempts to prefetch, the weaker the temporal correlation and consequently the more often the address mapping changes. Twilight's online metadata component improves its ability to adapt, which mitigates these effects.

A. *Comparison with Delta Prefetchers*

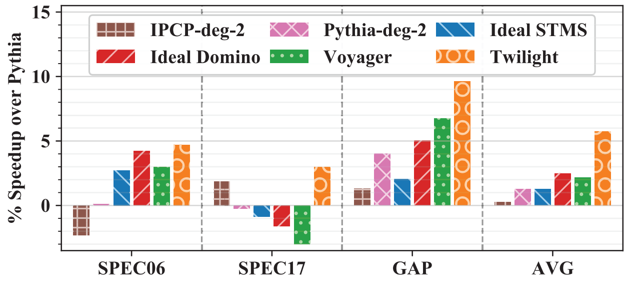
We now show that delta prefetchers largely target different access patterns from temporal prefetchers and that Twilight profitably hybridizes with them. To fairly compare predictive power, we evaluate both delta and temporal prefetchers in the LLC with degree 1 unless otherwise stated.

Figure 8a compares temporal prefetchers against delta prefetchers. We see that that Voyager (1.7%) and Twilight (5%) outperform IPCP (−0.9%) and Pythia, particularly on SPEC06 which is the most irregular. Moreover, we observe that only neural temporal prefetchers outperform Pythia and that only Twilight outperforms Pythia across benchmark suites.

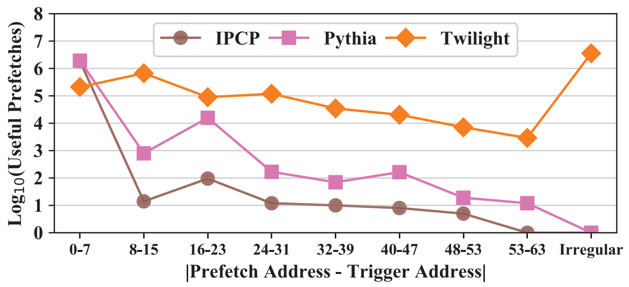
In Figure 8b, we hybridize various temporal prefetchers with Pythia-deg-1 and see that Twilight (5.8%) significantly outperforms Voyager (2.2%), Domino (2.5%), and STMS (1.3%). At equal prefetch degree, hybridized Twilight outperforms degree-2 delta prefetchers, IPCP-deg-2 and Pythia-deg-2, by 5.5% and 3.5% respectively, demonstrating that hybridizing across types of prefetchers provides better performance than just increasing degree.



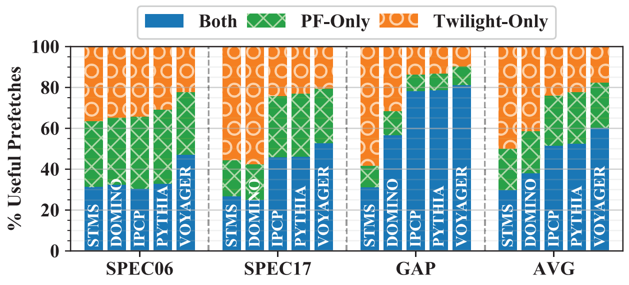
(a) Temporal vs Delta Prefetcher Performance



(b) Temporal Prefetchers Hybridized with Delta Prefetchers



(c) Prefetch Coverage Breakdown for xalancbmk



(d) Prefetch Coverage Overlap

Fig. 8: Temporal prefetchers are complementary to delta prefetchers. (a) Degree-1 performance. (b) Hybridized temporal prefetcher performance vs delta prefetchers with equal degree. (c) Prefetch coverage bucketed by the magnitude of the predicted delta for xalancbmk. (d) Breakdown of prefetch overlap between Twilight and baselines.

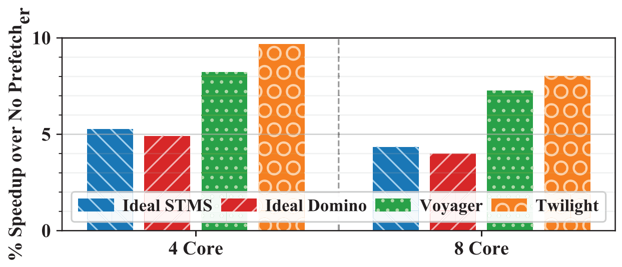


Fig. 9: Multi-Core Speedup across 50 4-core mixes and 75 8-core mixes of SPEC 2006 and GAP benchmarks.

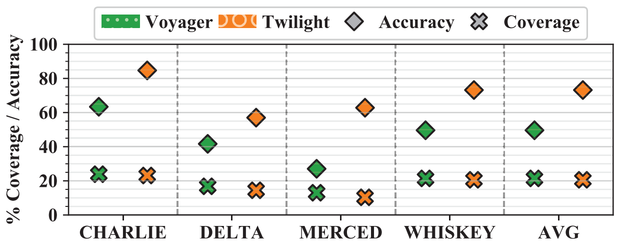


Fig. 10: Google Workloads Prefetch Accuracy and Coverage

Figure 8c breaks down, for xalancbmk, useful prefetches based on the distance between the prefetch and the trigger. Whereas IPCP and Pythia[4] are biased to small deltas, Twilight predicts deltas more uniformly, indicating that it prefetches more irregularly within pages. Furthermore, Twilight fruitfully prefetches across pages—which delta prefetchers cannot—and had more useful irregular prefetches (3.6M) than IPCP and Pythia had useful prefetches (1.9M). Figure 8d shows that this disjoint coverage occurs in all benchmark suites where 20 – 50% of useful prefetches can only be issued by Twilight.

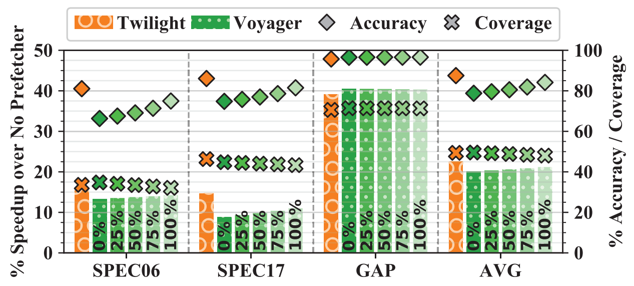
Overall, Twilight is capable of significantly boosting performance by accurately prefetching irregular accesses that delta prefetchers cannot cover. Moreover, it does so better than other neural and idealized non-neural temporal prefetchers.

B. Multi-Core Results

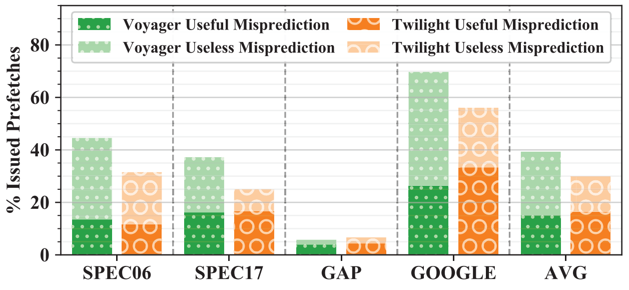
Figure 9 shows our multi-core results for 50 4-core mixes and 75 8-core mixes of random combinations of SimPoints sampled from the SPEC 2006, SPEC 2017, and GAP benchmarks suites. All cores are warmed up for 225M instructions and simulated for 75M instructions.

In these multi-core settings where available bandwidth is more scarce, the gap between the neural and non-neural prefetchers grows because of the higher accuracy of the neural prefetchers. On the 4-core mixes, Twilight (9.7%) and Voyager (8.2%) outperform both idealized Domino (5.3%) and idealized STMS (4.9%). This trend continues in the 8-core

[4]For Pythia, we add all possible deltas to its action set.



(a) Filter Voyager's Prefetches by Twilight's No-Prefetch



(b) Usefulness of Mispredicted Prefetches

Fig. 11: (a) We filter Voyager prefetches on trigger accesses for which Twilight predicts No-Prefetch, and we see that increased filtering improves Voyager's accuracy, showing that Twilight doesn't prefetch on accesses with weak temporal correlation. (b) Twilight issues fewer mispredicted prefetches (i.e. prefetches to addresses that aren't the next PC-localized access) than Voyager; moreover, Twilight's mispredictions are more often useful because candidate selection culls out irrelevant pages.

setting where Twilight (8%) and Voyager (7.3%) maintain a $3-4\%$ advantage over Domino (4.4%) and STMS (4%).

C. Google Results

Figure 10 compares Voyager with Twilight on the public Google server workloads. Twilight's higher adaptability enables it to achieve significantly higher accuracy (73.2%) than Voyager (49.6%) while seeing just 0.8% lower coverage. Moreover, the larger memory footprints from the server workloads increases the amount of information that Voyager must learn, which also contributes to its reduced prefetch accuracy.

D. Frequency-Based Candidate Selection

We now provide insights into the benefits of *frequency-based candidate selection,* and we show that Twilight performs more complicated prediction than just naively multiplying the frequency values.

1) Better Prefetch Accuracy: Frequency-based candidate selection improves Twilight's prefetch accuracy for two reasons: (1) Twilight can predict to not prefetch and (2) Twilight's mispredictions are more likely to be useful.

Figure 11a evaluates the utility of Twilight's No-Prefetch predictions in filtering out prefetches that Voyager does prefetch. We see that Voyager's accuracy monotonically improves with increased filtering, and at 100% filtering, Voyager trades off 1.5% coverage for an extra 5.4% accuracy and 1.1% speedup, showing that Twilight's No-Prefetch predictions are mostly for poorly correlated addresses. Unlike candidate selection where Twilight learns to not prefetch infrequent successors outside of its set of candidates, Voyager's problem formulation cannot determine when to not prefetch without it becoming some analog of candidate selection.

Figure 11b shows that Twilight's mispredictions are more likely to be useful than Voyager's. That is, even when a prefetcher fails to predict the intended target address (i.e. the address of the next access in the PC-localized stream), the prefetch can still be useful. We see that compared to Voyager, Twilight issues 9.6% fewer mispredicted prefetches.

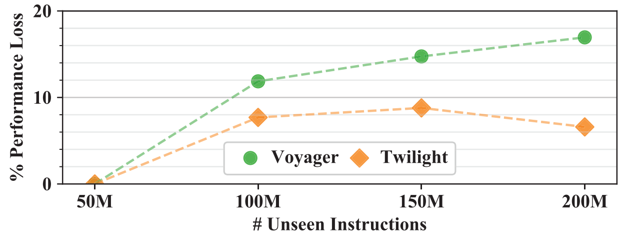


Fig. 12: Performance loss relative to speedup at 50M instructions. Whereas Voyager increasingly loses speedup in unseen code regions, Twilight adapts and generalizes better.

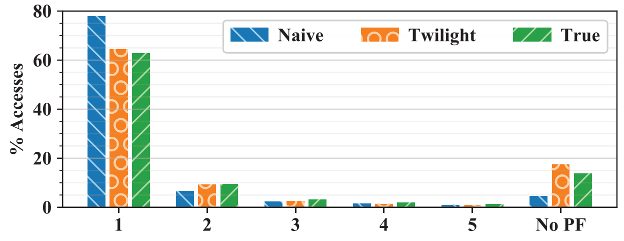


Fig. 13: Twilight's distribution of predicted candidates compared against the true distribution and the distribution from naively multiplying the DPFs together.

Moreover, Twilight's rate of useful mispredictions is 12.4% higher because Twilight's mispredictions still come from the set of prefetch candidates and therefore are strongly temporally correlated with the trigger address. Consequently, a Twilight misprediction is more likely to be accessed in the near future than a Voyager misprediction, which can be some arbitrary address from the entire address space.

2) Generalizability: Frequency-based candidate selection provides generalizability by enabling Twilight to prefetch addresses not in the training data. Figure 12 shows Twilight's and Voyager's performance throughout the evaluation region—

normalized to their speedups at 50M instructions. Voyager continually loses performance throughout evaluation, with 17% performance loss at 200M, while Twilight suffers only 6.6% performance loss and actually recovers performance between 150M and 200M instructions, demonstrating that Twilight's dynamic metadata collection improves its adaptability.

3) Predicted Candidate Distribution: Figure 13 shows that Twilight closely predicts the true distribution of successor candidates. The x-axis corresponds to the candidate index, and the y-axis is the percentage of accesses for which that candidate was chosen. The naive predictor multiplies the DPF values together to predict the most likely candidate.

Compared to the true distribution, the naive predictor overpredicts the top candidate by 15%, whereas Twilight overpredicts by just 1.5%. Similarly, for the other candidates, Twilight more closely mirrors the true distribution than the naive predictor. From the second candidate onward, both predictors underpredict the true distribution. Since these candidates occur less often, Twilight cannot easily learn these correlations and instead learns to not prefetch (3.7% more than the true distribution) rather than pollute the cache. By contrast, Voyager always prefetches even for unpredictable input.

VI. T-LITE Evaluation

Whereas Twilight was evaluated idealistically, our evaluation of T-LITE models storage costs and inference latency with higher fidelity to more fairly compare against Triage [52], a practical table-based temporal prefetcher.

A. DPF Metadata Management

T-LITE changes two hyperparameter settings to dramatically reduce the number and size of DPF metadata entries:

- T-LITE reduces the number of possible successors from $N = 20$ to $N = 4$, trading off 9.8% coverage (see Figure 2b) for a 80% reduction in DPF entry size and a 30% reduction in prediction latency.
- T-LITE only uses $f(P, 1)$, the DPF distribution for immediate page successors, reducing the number of metadata entries by two-thirds.

Since DPF metadata is tracked at a page granularity, it requires far fewer entries than other temporal prefetchers [8], [52], [53]. This observation combined with the above optimizations enables T-LITE to retain most of Twilight's performance with just 64 KB of DPF metadata. Unlike Triage which partitions out 1 MB of the last-level cache, T-LITE stores its DPF metadata in an on-chip metadata cache separate from the LLC (see Figure 14). We faithfully model this metadata cache as a 8-way, set-associative cache using LRU replacement.

B. Neural Model

1) Behavioral Clustering: As described in Section III-B, T-LITE utilizes *behavioral clustering* to produce a fixed number of cluster embeddings and therefore a constant model size. Figure 16a shows that this shrinks the model size from 10.5 MB (10.8× smaller than Voyager) to 3.2 MB (35.5×).

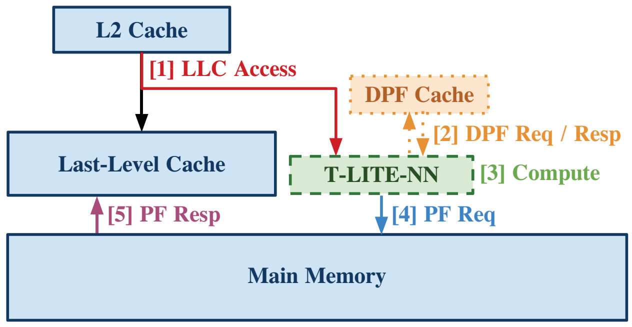


Fig. 14: T-LITE's Hardware Design. [1] On LLC access, [2] T-LITE fetches the DPF metadata from the DPF cache. [3] T-LITE computes the prefetch address and [4] issues the prefetch request to DRAM which [5] fills in the LLC.

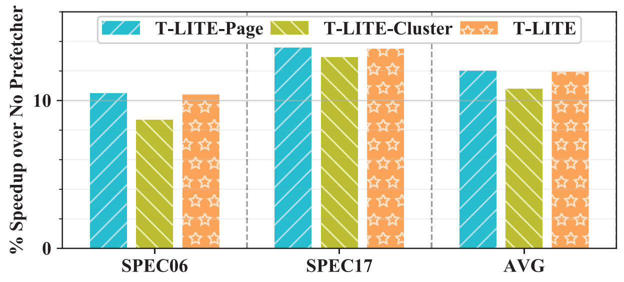


Fig. 15: With *behavioral clustering*, a fine-tuning pass recalibrates T-LITE to take clusters as inputs instead of pages.

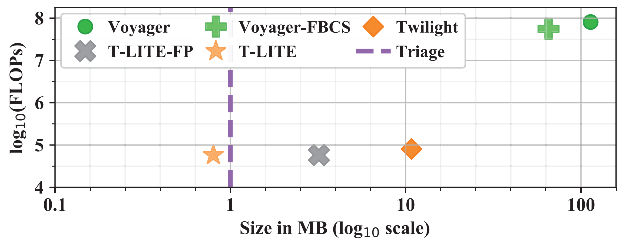
Compared to the page-based T-LITE-Page, Figure 15 shows that utilizing clusters without fine-tuning (T-LITE-Cluster) drops speedup by 1.3% while fine-tuning (T-LITE) reduces this drop to 0.1%, demonstrating the importance of recalibrating the neural model to take clusters as input instead of pages.

2) Weight Quantization: We further reduce our neural model size by quantizing the model weights from 32-bit floating point numbers (T-LITE-FP) to 8-bit integers (T-LITE), reducing the model size to 0.8 MB (142× reduction) with zero performance loss. Figure 16b shows that further quantization is possible where T-LITE-4bit outperforms Triage by 3.2% with a 0.4 MB (284× reduction) neural model.

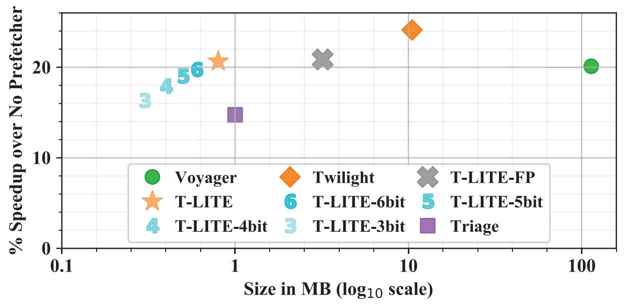
C. Speculative Prediction

After quantization, T-LITE only requires 57K IOPs per prediction (1421× faster than Voyager). Given matrix extensions such as Intel AMX [4] or Arm SME [51], which can perform 8-bit integer operations at 2048 IOPs/cycle, T-LITE's latency is approximately 29 cycles.

However, since T-LITE has high prediction accuracy, it can speculatively predict ahead of time based on its own prefetches. By doing so, T-LITE can have zero-cycle prefetch latency when the prior prefetch is correct as the latency between LLC accesses is larger than T-LITE's prediction latency. Thus, we model T-LITE to have a 29 cycle delay if the prior prefetch was incorrect and a 0 cycle delay otherwise.



(a) Storage vs Latency



(b) Storage vs Performance

Fig. 16: (a) Our abstractions provide several orders of magnitude improvement over Voyager. Voyager-FBCS utilizes candidate selection and reduces storage by 42% and latency by 32%. (b) All quantized versions of T-LITE outperform Triage across performance and storage.

D. Results

Figure 17 shows that T-LITE achieves a 20.7% speedup, outperforming Triage (14.8%), a practical temporal prefetcher. Moreover, T-LITE also achieves 2.6% higher accuracy and 7.8% higher coverage. T-LITE outperforms Triage on both SPEC 2017 and GAP, but it underperforms on SPEC 2006 by 2%. As noted in Section V, SPEC 2006 has workloads with wider successor distributions, and because T-LITE further reduces the number of successors that it tracks, it has 9% lower prefetch coverage but maintains similar accuracy to Triage.

Compared to Triage's 1 MB metadata storage, all quantized versions of T-LITE require less total storage: 64 KB of metadata and 0.3 to 0.8 MB of neural weights. Moreover, T-LITE's 29 cycle prediction latency is similar to the 20 cycle prefetch delay that Triage would incur to read its metadata from the LLC. Finally, because its DPF metadata is stored outside of the LLC, T-LITE does not contend with LLC demand accesses, while Triage's metadata accesses do.

Figure 16b shows that T-LITE matches Voyager's performance despite the many orders of magnitude improvement in storage and latency. Moreover, even with further reduced precision, T-LITE-6bit still matches Voyager's performance.

E. Transfer Learning

We now evaluate T-LITE's ability to transfer what its learned across program inputs. We first show that unseen pages can be accurately mapped to their corresponding behavioral clusters. We then use this mapping scheme to train T-LITE on one input and evaluate it on another.

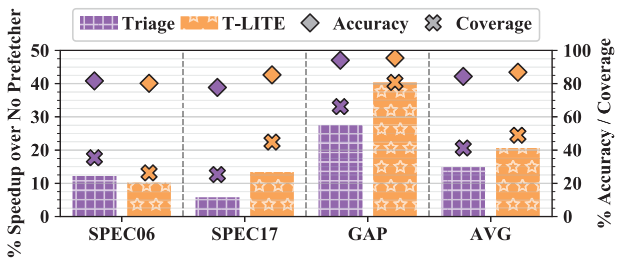


Fig. 17: T-LITE vs Triage

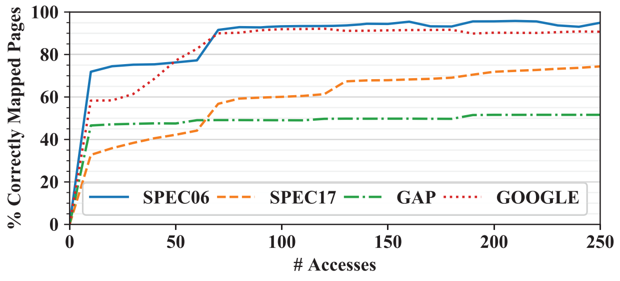


Fig. 18: Dynamically mapping pages to clusters during runtime by assigning the cluster with the most similar offset transition distribution yields high accuracy with few accesses.

1) Dynamic Page-to-Cluster Mapping: Figure 4 demonstrates that similar page embeddings have similar offset transition matrices. Given this relationship, we show in Figure 18 that we can dynamically map unseen pages to their behavioral clusters with reasonable accuracy. For each cluster, we aggregate an offset transition matrix across the pages assigned to that cluster. During runtime, we track the offset transitions for unseen pages and assign these pages to the cluster whose aggregate offset transitions is most similar. The mapping accuracy increases with each access to the page and for SPEC 2006, the accuracy reaches 95.6% at 250 accesses. Given the large number of clusters (4096), this simple scheme achieves decent accuracy with cheaper overhead than dynamically training a page embedding for every new page.

2) Cross Input Evaluation: We now evaluate T-LITE's transferability across different program inputs. Specifically, we evaluate performance on the GAP benchmark suite with input graphs from the SuiteSparse collection [16] across different graph types: web crawls, road networks, and citation networks.

Since there are 3 graph types, we can select one for training (A), one for validation (B), and one for evaluation (C). First, we train T-LITE on every graph from A and select the model α that performs the best on B. We normalize α's performance on B by dividing it by the average performance of the other A-models on B. We repeat this process with A and B switched to produce another best model β and use the normalized scores to select whether to transfer α or β to C, ensuring that the

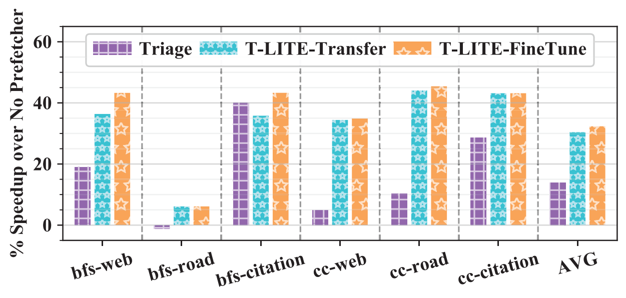


Fig. 19: T-LITE transfers well on GAP across graph inputs from different domains (web, road, citation).

chosen model has no exposure to the evaluation data.

Figure 19 shows that T-LITE-Transfer significantly outperforms Triage (30.4% speedup vs. 13.8% speedup) despite never having trained on the evaluated input graph domain. Moreover, compared to T-LITE-FineTune which is trained on the actual evaluation data, T-LITE-Transfer is able to achieve 94% of T-LITE-FineTune's performance, demonstrating how well the dynamic mapping scheme is able to align new pages to behavioral clusters. Figure 18 shows low top-1 page-to-cluster mapping accuracy for GAP, but these results indicate that even if the dynamically mapped cluster isn't the best cluster, the mapped behavioral cluster still performs well because it has sufficiently similar offset transition behavior.

F. T-LITE's Limitations

While T-LITE's latency and storage is feasible, inference would cost more energy than the savings provided by superior prefetching. Moreover, despite T-LITE profitably transferring knowledge learned across different program inputs, it's still constrained to the program it was trained on and requires significant offline-training on more representative data to further improve its generalization.

VII. Future Work

Our reformulation of temporal prefetching bridges most of the gap between the prior state-of-the-art, Voyager, and the actual hardware deployment of a neural temporal prefetcher. Moreover, it provides various avenues for future work in temporal prefetching.

A. Lightweight Neural Temporal Prefetching

We believe frequency-based candidate selection is a viable path forward for improving the accuracy of existing temporal prefetchers. We can envision a lightweight neural model like a perceptron accompanying a temporal prefetcher to decide which successor prefetch candidate should be issued given features such as the PC, the access history, bandwidth, etc.

B. Insights for Non-Neural Temporal Prefetching

Voyager's page and offset split make it feasible to train and evaluate neural temporal prefetchers. In tracking our page-granularity DPF metadata, we observe a significant reduction in the amount of required metadata for our temporal prefetcher. This decomposition of addresses could likewise help future temporal prefetchers reduce their metadata by utilizing page-to-page temporal locality.

VIII. Conclusions

The Voyager prefetcher showed the tremendous promise that neural networks hold for performing temporal data prefetching. Unfortunately, its costs in terms of model size, prediction latency, and training time are so large that they cannot be bridged by standard machine learning techniques such as distillation, pruning, and quantization. Moreover, the direct use of addresses in temporal prefetching precludes the use of any kind of staged deployment, in which the neural prefetcher is trained offline with predictions being performed online.

In this paper, we have reformulated the temporal prefetching problem to make it fundamentally more suitable for practical implementation in hardware. By introducing two novel layers of indirection that abstract a temporal prefetcher away from specific data addresses, we have enabled neural prefetchers to operate in a staged manner, where the neural model is first trained offline on representative program traces and then deployed online, where it can dynamically track address metadata to allow it to adapt to different program phases and to prefetch unseen addresses across program inputs.

Our new formulation also dramatically reduces the size and cost of the neural model. Compared to Voyager, our Twilight neural prefetcher utilizes *frequency-based candidate selection* to improve the storage (10.8× smaller) and latency (988× faster) by orders of magnitude, while also generalizing better, allowing it to outperform Voyager by 4%.

We have also introduced T-LITE, which utilizes *behavioral clustering*, quantization, and other optimizations to move neural temporal prefetching towards feasible hardware deployment by trading off performance. T-LITE outperforms Triage by 5.9% and even matches Voyager's performance while having 142× less storage and 1421× faster prediction latency. We have evaluated versions of T-LITE whose model size ranges from 0.3 MB to 0.8 MB and all outperform Triage provisioned with 1 MB for on-chip metadata storage. These versions of T-LITE require just 64 KB of on-chip storage for metadata, demonstrating that the amount of dynamic metadata for temporal prefetching can be orders of magnitude smaller than previously thought because a significant amount of prefetching knowledge can be baked into the neural model.

Acknowledgments

We thank the anonymous reviewers, Daniel Jiménez, and members of the Speedway Research group for their helpful comments on earlier drafts of this paper. This work was funded in part by a gift from Arm Inc., NSF Grant CCF-1823546, and a gift from Intel Corporation through the NSF/Intel Partnership on Foundational Microarchitecture Research.

REFERENCES

[1] "Spec 2006," 2006. [Online]. Available: https://www.spec.org/cpu2006/

[2] "Spec 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[3] "Sunny cove microarchitecture: Going deeper and wider," 2019. [Online]. Available: https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3

[4] "Accelerate artificial intelligence (ai) workloads with intel advanced matrix extensions (intel amx)," 2022. [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-12/accelerate-ai-with-amx-sb.pdf

[5] "Google workload traces," 2022. [Online]. Available: https://dynamorio.org/google_workload_traces.html

[6] A. Asgari, A. Gunter, M. Saeidi, M. Lis, and P. Nair, "MPMLP: A Case for Multi-Page Multi-Layer Perceptron Prefetcher," 2021.

[7] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:11212020

[8] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 131–142.

[9] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 399–411.

[10] S. Beamer, K. Asanovic, and D. Patterson, "The gap benchmark suite," 2017.

[11] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadati, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," 2022.

[12] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, oct 2021. [Online]. Available: https://doi.org/10.1145%2F3466752.3480114

[13] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 113. [Online]. Available: https://doi.org/10.1145/3307650.3322207

[14] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 133144. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1145/2151024.2151043

[15] Y. Chou, "Low-cost epoch-based correlation prefetching for commercial applications," in *MICRO*, 2007, pp. 301–313.

[16] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[17] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, p. 3748. [Online]. Available: https://doi.org/10.1145/2150976.2150982

[18] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "OPTQ: Accurate quantization for generative pre-trained transformers," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=tcbBPnfwxS

[19] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jiménez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," 2022.

[20] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *J. Instr. Level Parallelism*, vol. 7, 2005.

[21] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1919–1928.

[22] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015.

[23] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[24] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural Computation*, vol. 3, no. 1, pp. 79–87, 1991.

[25] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 247259. [Online]. Available: https://doi.org/10.1145/2540708.2540730

[26] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, "Apt-get: Profile-guided timely software prefetching," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 747764. [Online]. Available: https://doi.org/10.1145/3492321.3519583

[27] D. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206.

[28] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 252–263.

[29] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[30] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML'20. JMLR.org, 2020.

[31] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.

[32] A. Navarro-Torres, B. Panda, J. Alastruey-Bened, P. Ibez, V. Vials-Yfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 975–991.

[33] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, 2005.

[34] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.

[35] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. Portland Oregon: ACM, Jun. 2015, pp. 285–297. [Online]. Available: https://dl.acm.org/doi/10.1145/2749469.2749473

[36] 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, oct 2019. [Online]. Available: https://doi.org/10.1145/3345000

[37] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 291–303.

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

[39] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 141–152.

[40] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying Deep Learning to the Cache Replacement Problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 413–425. [Online]. Available: https://dl.acm.org/doi/10.1145/3352460.3358319

[41] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of*

the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 861–873.

[42] Y. Solihin, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 171–182.

[43] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 6980, jun 2009. [Online]. Available: https://doi.org/10.1145/1555815.1555766

[44] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ISCA*, 2009, pp. 69–80.

[45] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting memory accesses: the road to compact ML-driven prefetcher," in *Proceedings of the International Symposium on Memory Systems*. Washington District of Columbia USA: ACM, Sep. 2019, pp. 461–470. [Online]. Available: https://dl.acm.org/doi/10.1145/3357526.3357549

[46] A. Srivastava, T.-Y. Wang, P. Zhang, C. A. F. De Rose, R. Kannan, and V. K. Prasanna, "MemMAP: Compact and Generalizable Meta-LSTM Models for Memory Access Prediction," in *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science, H. W. Lauw, R. C.-W. Wong, A. Ntoulas, E.-P. Lim, S.-K. Ng, and S. J. Pan, Eds. Cham: Springer International Publishing, 2020, pp. 57–68.

[47] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," 2019.

[48] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Taipei, Taiwan: IEEE, Oct. 2016, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/7783705/

[49] L. van der Maaten and G. E. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[50] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 79–90.

[51] F. Wilkinson and S. McIntosh-Smith, "An initial evaluation of arms scalable matrix extension," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2022, pp. 135–140.

[52] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 9961008. [Online]. Available: https://doi.org/10.1145/3352460.3358300

[53] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 1–13.

[54] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 118–130.

[55] P. Zhang, R. Kannan, A. Nori, and V. Prasanna, "A2p: Attention-based memory access prediction for graph analytics [a2p: Attention-based memory access prediction for graph analytics]," *the 11th International Conference on Data Science, Technology and Applications*. [Online]. Available: https://par.nsf.gov/biblio/10376313

[56] P. Zhang, R. Kannan, X. Tong, A. V. Nori, and V. K. Prasanna, "Sharp: Software hint-assisted memory access prediction for graph analytics," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–8.

[57] P. Zhang, A. Srivastava, B. Brooks, R. Kannan, and V. K. Prasanna, "RAOP: Recurrent Neural Network Augmented Offset Prefetcher," in *The International Symposium on Memory Systems*. Washington DC USA: ACM, Sep. 2020, pp. 352–362. [Online]. Available: https://dl.acm.org/doi/10.1145/3422575.3422807

[58] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Transformap: Transformer for memory access prediction," in *The International Symposium on Computer Architecture (ISCA), ML for Computer Architecture and Systems Workshop*, 2021.

[59] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation for attention-based variable-degree prefetching," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, ser. CF '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 103112. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3528416.3530236

[60] P. Zhang, A. Srivastava, T.-Y. Wang, C. A. F. De Rose, R. Kannan, and V. K. Prasanna, "C-MemMAP: clustering-driven compact, adaptable, and generalizable meta-LSTM models for memory access prediction," *International Journal of Data Science and Analytics*, vol. 13, no. 1, pp. 3–16, Jan. 2022. [Online]. Available: https://doi.org/10.1007/s41060-021-00268-y