Fine-Grained Address Segmentation for Attention-Based Variable-Degree Prefetching

Pengmiao Zhang
University of Southern California
Los Angeles, California, USA
pengmiao@usc.edu

Ajitesh Srivastava
University of Southern California
Los Angeles, California, USA
ajiteshs@usc.edu

Anant V. Nori
Processor Architecture Research Lab,
Intel Labs
Bangalore, India
anant.v.nori@intel.com

Rajgopal Kannan
US Army Research Lab-West
Los Angeles, California, USA
rajgopal.kannan.civ@army.mil

Viktor K. Prasanna
University of Southern California
Los Angeles, California, USA
prasanna@usc.edu

ABSTRACT

Machine learning algorithms have shown potential to improve prefetching performance by accurately predicting future memory accesses. Existing approaches are based on the modeling of text prediction, considering prefetching as a classification problem for sequence prediction. However, the vast and sparse memory address space leads to large vocabulary, which makes this modeling impractical. The number and order of outputs for multiple cache line prefetching are also fundamentally different from text prediction.

We propose TransFetch, a novel way to model prefetching. To reduce vocabulary size, we use fine-grained address segmentation as input. To predict unordered sets of future addresses, we use delta bitmaps for multiple outputs. We apply an attention-based network to learn the mapping between input and output. Prediction experiments demonstrate that address segmentation achieves 26% - 36% higher F1-score than delta inputs and 15% - 24% higher F1-score than page & offset inputs for SPEC 2006, SPEC 2017, and GAP benchmarks. Simulation results show that TransFetch achieves 38.75% IPC improvement compared with no prefetching, outperforming the best-performing rule-based prefetcher BOP by 10.44% and ML-based prefetcher Voyager by 6.64%.

CCS CONCEPTS

• **Computer systems organization** → **Processors and memory architectures**; **Neural networks**; • **Information systems** → Data mining.

KEYWORDS

prefetching, machine learning, attention, address segmentation

ACM Reference Format:
Pengmiao Zhang, Ajitesh Srivastava, Anant V. Nori, Rajgopal Kannan, and Viktor K. Prasanna. 2022. Fine-Grained Address Segmentation for Attention-Based Variable-Degree Prefetching. In *19th ACM International Conference on Computing Frontiers (CF'22), May 17–19, 2022, Torino, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3528416.3530236

CF'22, May 17–19, 2022, Torino, Italy

ACM ISBN 978-1-4503-9338-6/22/05.
https://doi.org/10.1145/3528416.3530236

1 INTRODUCTION

Memory latency is becoming an overwhelming bottleneck in computer performance due to the "memory wall" [4, 72] problem, especially with the advent of GPUs [43], TPUs [28], and heterogeneous architectures [17, 52] that accelerate computation. Prefetching is critical in reducing program execution time and improving instructions per cycle (IPC) by hiding the latency. It looks at patterns of memory accesses sequences and uses the past information to forecast the near future accesses so as to start fetching the data before the miss occurs [13, 65]. Existing prefetchers are mainly heuristic, predicting via pre-defined rules, based on the observation from the locality of references [1, 6–9, 14, 19, 20, 22, 23, 25–27, 32, 34, 37, 42, 44, 49, 55, 57–59, 69–71]. With the rise of new workloads, such as graph analytics [2, 38, 54], data mining [29, 63], and AI applications [21, 64, 67], rule-based prefetchers are not powerful enough to adapt to the increasingly irregular, indirect, and complex memory access patterns.

Machine learning-based data prefetchers are gaining increasing attention to pursue higher performance for memory access prediction [15, 16, 61] and prefetching [3, 45, 46, 56, 61, 73]. Prefetching is commonly modeled as classification for sequence prediction [3, 15, 16, 46, 56, 61, 74, 75], which is analogous to the problem setting of text prediction [15] in natural language processing (NLP) [39]. However, this analogy is not perfect. First, the unique memory addresses for an application can be tens of millions, which is orders of magnitude larger than natural language vocabulary and exceeds the capability of machine learning models. This problem is known as *class explosion* [56]. Existing approaches partly solve this problem by working on memory access address deltas [15, 16, 61] or splitting an address by page and offset [56]. Second, *tokenization* [68], as a commonly used technique in NLP that maps a meaningful word into nonsensitive numerical data for model processing, is also borrowed by existing ML-based prefetching models for preprocessing. Tokenization results in extra storage to save the mapping tables (token dictionaries) in hardware implementation, but the cost has been neglected by previous works [3, 15, 16, 46, 56, 61, 74, 75]. Third, unlike text prediction

with a ground truth of future words in a fixed order, in prefetching there is no ground truth of a certain memory address that should be prefetched. This is know as the *labeling* problem [56]. Any access address following the current access could be a potential label. For a prefetcher that can prefetch multiple blocks for each trigger (prefetch degree > 1), the order of predicted block addresses for one prefetch is also insignificant. Lastly, the *latency* overhead of ML-based prefetcher is also ignored under this setting. LSTM (Long short-term memory) [18] is a commonly used prediction model [16, 56, 61, 62, 74, 75] due to its advantage in sequence modeling. However, the recurrent structure of LSTM is hard to be parallelized and the inference latency increases linearly with the input time step length. Recent success of attention-based models, e.g. the Transformer [66], provides insight into solving this problem by the virtue of high parallelizability.

To solve the problem of *class explosion, tokenization, labeling*, and *latency*, we propose TransFetch (Transformer for prefetching), an attention-based prefetcher that supports variable-degree prefetching. We model prefetching as a multi-label classification problem. To overcome class explosion for memory address input, we propose an address segmentation method to reduce the vocabulary without information loss. It avoids tokenization as the processed value can be directly fed into a neural network. For labeling, we use delta bitmaps that collects unordered sets of future deltas to the current address, which paves the way for multiple block (cache line) prefetching. In inference, an optimal threshold that maximizes the F1-score is adapted to adjust the prefetch degrees (the number of blocks to be prefetched) and balance prefetch aggressiveness. We apply a powerful and embarrassingly parallelizable attention-based network to learn the mapping between the input segmented addresses and the delta bitmap labels. The model also supports incorporation of more context features (program counters, page distances) to enhance the prediction performance. Besides, we further offset the model inference latency by artificially introducing estimated latency in training and then performing distance prefetching.

Overall, our main contributions are:

- We propose TransFetch, an ML-based prefetcher that models prefetching as multi-label classification. Our model uses address segmentation for input, delta bitmap for labeling, attention-based network with context enhancement for prediction, optimal-threshold confidence throttling mechanism for variable-degree prefetching, and distance prefetching for hiding inference latency.
- We demonstrate the effectiveness of address segmentation, attention-based network, and context enhancement in prediction experiments. Results show that address segmentation achieves 26% - 36% higher F1-score than delta inputs and 15%-24% higher F1-score than page & offset inputs. Attention-based model achieves 10% - 13% higher F1-score than LSTM and Temporal Convolutional Networks (TCN) [33]. Context enhancement raises the F1-score by 3.1% - 9.1%.
- We evaluate the performance of TransFetch using accuracy, coverage, and IPC improvement. Results show that our method achieves 88.56% prefetch accuracy and 60.54% prefetch coverage. It improves IPC by 38.75% compared with no prefetching, outperforming the best-performing rule-based prefetcher BOP by 10.44%, and ML-based prefetcher Voyager by 6.64%.

2 BACKGROUND AND RELATED WORK

In this section, we provide background for data prefetching, attention mechanism, along with the related prior works.

2.1 Data Prefetching

A prefetching process is a form of speculation that aims to predict the future data addresses and fetch the data before it is needed. Prefetch degree is the number of fetching blocks for each prefetching operation, which indicates the aggressiveness of a prefetcher. While a higher degree is likely to bring more useful data into cache, it may introduce cache pollution due to wrong predictions.

Rule-based prefetching. Traditional prefetchers learn from predefined rules, usually exploiting spatial or temporal localities. For example, Spatial Memory Streaming (SMS) [59] prefetcher identifies code-correlated spatial patterns to predict future accesses. Spatial prefetcher BOP [37] and VLDP [55] learn from history access page offsets or deltas and predict future accesses within a spatial region. Temporal prefetchers like Irregular Stream Buffer (ISB) [23] and Domino [1] predict temporally correlated memory accesses by recording and replaying the history access sequences. Most rule-based prefetchers require manually configured prefetch degree [1, 23, 37, 55]. Signature Path Prefetcher (SPP) [30] uses a path confidence-based lookahead mechanism to balance the prefetching aggressiveness and achieves variable-degree prefetching.

ML-based prefetching. Several prior works have explored the application of machine learning on data prefetching. Rahman et al. [50] use logistic regression and decision tree for pattern classification. Hashemi et al. [16] present an extensive evaluation of LSTM in learning memory access patterns. Some other works [3, 46, 73] also demonstrate the effectiveness of LSTM in memory access prediction. Srivastava et al. [61, 62] use compact LSTM to address the class explosion problem. RAOP [74] leverages LSTM-based models for virtual address predictions. C-MemMAP [75] combines clustering and meta-models to reduce the model size. Seq2seq modeling [40] based on LSTM encoder-decoder structure has been applied for memory sequence prediction. Shi et al. [56] propose Voyager that predicts both page sequence and page offsets using two LSTM models along with a dot-product attention mechanism. Existing ML-based prefetchers use history memory access sequence to predict the next memory access address [3, 16, 46, 56, 61, 62, 73–75], which leads to a prefetch degree as one. These models require recurrent greedy/beam search or accepting low-probability candidates to realize higher degree prefetching.

2.2 Attention

Transformer [66] suggested a sequence model based on multi-head attention mechanism and feed-forward network, dispensing with recurrent structures.

Self-attention. Self-attention takes the embedding of items as input, converts them to three matrices through linear projection,

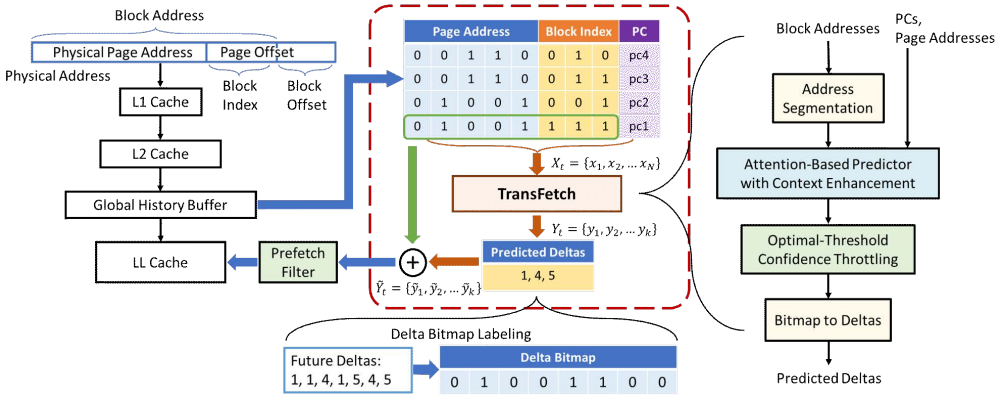


Figure 1: Overall architecture of TransFetch. We have an input sequence of history memory accesses $X_t = \{x_1, x_2, ..., x_N\}$ and output a set of desired block deltas $Y_t = \{y_1, y_2, ..., y_k\}$ to the current address. The final block address predictions $\widetilde{Y_t}$ are the addition of the current block address and the predicted deltas.

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, d the dimension of layer input.

Multi-head attention. One self-attention operation can be considered as one "head", we can apply multi-head self-attention (MSA) operation as follows:

$$\begin{aligned} \text{MSA}(Q, K, V) &= \text{Concat}\left(\text{head}_1, \ldots, \text{head}_\text{H}\right) W^O \\ \text{head}_\text{i} &= \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned} \tag{2}$$

where the projection matrics $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d\times d}$ and H is the number of heads.

Point-wise feed-forward. Point-wise feed-forward network (FFN) is defined as follows:

$$\text{FFN}(x) = \max\left(0, xW_1 + b_1\right)W_2 + b_2 \tag{3}$$

Previous ML-based prefetchers widely use recurrent neural networks (mainly LSTM) [3, 16, 46, 56, 61, 62, 73–75]. However, the recurrent structure of RNNs makes the model less practical due to high inference latency. A virtue of attention-based network is high parallelizability. Without recurrent steps, all input positions, hidden representations, and output dimensions can be computed in parallel [36]. In this work, we will explore using only attention-based networks suggested by [66] as a predictor for data prefetching.

3 APPROACH

In this section we describe TransFetch, an attention-based prefetcher that uses segmented address as input and achieves variable-degree prefetching through delta bitmap labeling. We formulate the memory access prediction task as a multi-label classification problem and design a neural model to fit the mapping. Since a prefetch must be in the unit of a block, we can consider only the block address space, the address configuration is shown in the left top of Figure 1.

Problem Formulation. Let $A_t = \{a_1, a_2, ..., a_N\}$ be the sequence of N history block addresses at time t, where $a_t = \{b_t^1, b_t^2, ..., b_t^p, ..., b_t^{p+c}\}$ represents the block address in binary with p-bit page address and c-bit block (cache line) index at time t. Let $PC_t = \{pc_1, pc_2, ..., pc_N\}$ be the sequence of N history program counters at time t. Let $X_t = \{(a_1, pc_1), (a_2, pc_2)..., (a_N, pc_N)\}$ be the input of the prediction model. Let $Y_t = \{y_1, y_2, ..., y_k\}$ be the set of k outputs associated with the unordered future k block deltas to the current block address. Our goal is to construct meaningful X_t to Y_t that are helpful in data prefetching. The final address predictions $\widetilde{Y_t} = \{\widetilde{y_1}, \widetilde{y_2}, ..., \widetilde{y_k}\}$ are the addition of current block address and the predicted deltas.

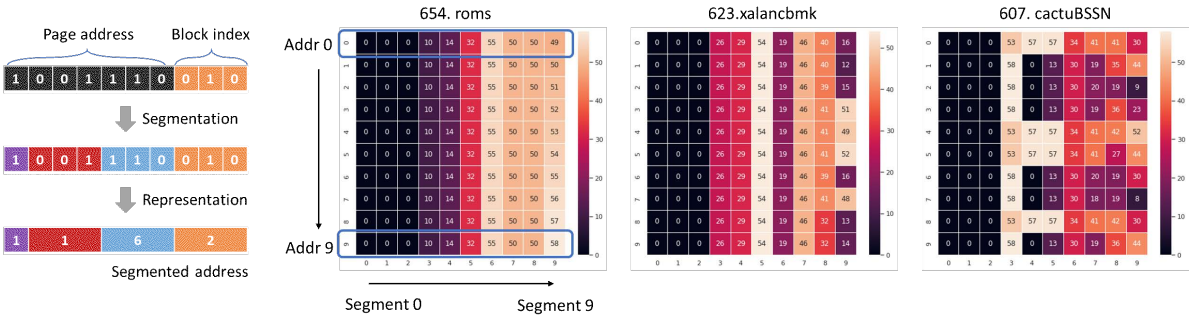
3.1 Overview of TransFetch

Figure 1 illustrates the overall architecture of TransFetch and how the model is applied in a hardware system. History block addresses are processed using address segmentation for model inputs, which solves *class explosion* and avoids *tokenization*, as is described in Section 3.2. As a solution for *labeling*, we take future deltas in the form of delta bitmap as training labels. In inference, optimal thresholds for output bitmaps are adapted to adjust the number of outputs (prefetch degree), as in Section 3.3. To reduce inference *latency*, a powerful and parallelizable attention-based network is designed for learning the mapping between input and output, as is described in Section 3.4. To further offset the latency, a distance prefetching scheme is discussed in Section 3.5.

3.2 Address Segmentation

We propose a simple approach called address segmentation (AS) to solve the class explosion problem in memory access prediction, keeping all the information in an address and avoiding tokenization.

Considering a block address with p-bit page address and c-bit block index, we can split this block address to $S = \lceil \frac{p+c}{s} \rceil$ segments, each with s bits. In this way, each segment can be represented in an integer within $[0 - 2^s)$. This range can be tuned appropriately for direct model input. One address then can be represented as a vector in dimension S.



(a) Address segmentation

(b) Examples of memory access patterns visualized in segmented addresses.

Figure 2: Address segmentation approach and memory access pattern cases visualized under segmented addresses.

There are two special cases. The first is when $s = 1$, the model input is a binary of an address. This case reveals the detail of an address in the highest granularity but requires a model to learn the correlation of each bit. The other case is when $s = c$, which means using the block size c as the segmentation basis and split the block address to $S = \lceil \frac{p}{c} \rceil + 1$ segments, the segment vocabulary is 2^c, 64 for $c = 6$. This case keeps the feature of internal page patterns and reduces the input dimension compared with binary inputs.

Figure 2a illustrates address segmentation for case $s = c$. Figure 2b visualizes three example pieces of memory access sequences from SPEC CPU 2017 [10] in form of this case, which illustrates the advantages of AS for preprocessing. While in application *654.roms*, the pattern lies mainly within a page (the column of segment 9), *623.xalancbmk* shows memory access skipping beyond the page limit. Furthermore, *607.cactuBSSN* shows more complex patterns, e.g., skippings between pages. AS keeps the information of an absolute address compared with solely delta, offset, or page inputs. In addition, AS avoids token dictionaries, saves storage space, and can process unknown input classes.

3.3 Variable-Degree Prefetching

3.3.1 Delta Bitmap Labeling. We use delta bitmaps as the format of labels and outputs. We aim to predict multiple unordered future deltas to the current block, as is shown in the bottom part of Figure 1. The labels are acquired from offline traces for training. First, future deltas $\mathbf{y_d}$ are collected within a look forward window W. Then, a delta bitmap at size B is filled to label the appearance of deltas by an arbitrary mapping rule $f : \mathbf{y_d} \rightarrow \mathbf{y_b}$, where $\mathbf{y_b}$ is the labeled bitmap, which can be used for multi-label model training. By designing the delta bitmap size B to be larger than a page, our model can learn and predict inter-page patterns, which addresses the weakness rule-based spatial prefetchers like BOP [37] and VLDP [55].

3.3.2 Optimal-Threshold Confidence Throttling. A neural network can be designed to output the probability of each bit being positive in a bitmap. We define this probability as *prefetch confidence* for the corresponding deltas in bitmap. Instead of using a fixed threshold, e.g. 0.5, to binarize the model output, we find the optimal threshold that maximizes the F1-score [53] in the step of model validation, between model training and testing. In inference, the output vector of prefetch confidence can be binarized using the optimal threshold, which forms the output delta bitmap. Then, the inverse mapping f^{-1} converts the output delta bitmap to the predicted delta vectors.

Using delta bitmap and optimal-threshold confidence throttling together for inference, our approach enables variable-degree prefetching: a model can predict and request prefetching variable number of future cache lines in one inference step.

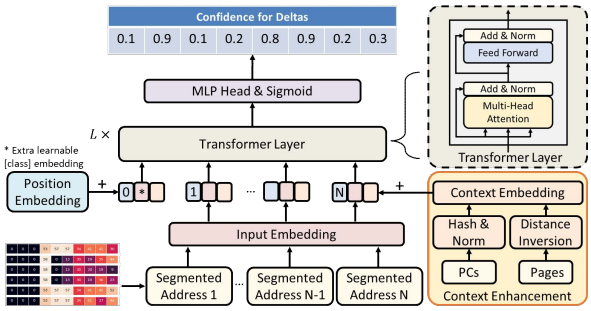


Figure 3: Attention-based memory access predictor with context enhancement.

3.4 Attention-Based Predictor

With the above well-defined input and output, we design an attention-based network to learn the mapping from the input segmented addresses to the delta bitmap labels. The model structure is depicted in Figure 3. First, we describe the basic model. Then we introduce context enhancement that utilizes more context information to boost the model performance.

3.4.1 Basic Model. Our model input is a 2D sequence of segmented addresses: $\mathbf{a}_S = [\mathbf{a}_S^1; \mathbf{a}_S^2; ...; \mathbf{a}_S^N] \in \mathbb{R}^{N \times S}$ where N is the number of history addresses and S is the dimension of a segmented address.

We flatten the input sequence and map to D dimensions using an input embedding layer, where D is the hidden dimension of Transformer layers. Inspired by BERT [11] and ViT [12], a trainable "classification token" denoted as $\mathbf{x}_{cls}$ is prepended to the input sequence, whose state is a comprehensive representation of the input sequence. We apply learnable 1D position embeddings [12]

to incorporate temporal information to input vector. The addition of input embeddings and position embeddings are fed into a the Transformer layers, which can be expressed as:

$$\mathbf{z}_0 = \left[\mathbf{x}_{cls}; \mathbf{a}_S^1\mathbf{E}; \mathbf{a}_S^2\mathbf{E}; \cdots; \mathbf{a}_S^N\mathbf{E}\right] + \mathbf{E}_{\text{pos}} \tag{4}$$

where $\mathbf{z}_0$ represents input sequence to the Transformer layers, $\mathbf{E}$ is the input embedding and $\mathbf{E}_{\text{pos}}$ is the position embedding.

The Transformer layer is based on multi-head attention and feed-forward network as described in Section 2.2. The output of L stacked Transformer layers will be fed into a multi-layer perceptron (MLP) head for multi-label classification, which is the same dimension as the delta bitmap size B. A sigmoid activation is applied to each output dimension and outputs the probability of this dimension being positive, which we use as the prefetch confidence for deltas.

3.4.2 Context Enhancement. The model can incorporate richer input features to boost the model performance using the same method as position embedding.

First, we incorporate program counters (PC), which are commonly used to help detecting memory patterns [23, 59]. To avoid tokenization, we use a *HASH_BITS* bit length folding method [35] as the hash function to compress the PC value pc_n. The hashed value is divided by 2^{HASH_BITS} for normalization, the processed PC input vector is $\mathbf{c_{pc}} = [c_{pc}^1; c_{pc}^2; ...; c_{pc}^N]$, c_{pc}^n is defined as:

$$c_{pc}^n = \frac{hash(pc_n)}{2^{HASH\_BITS}} \tag{5}$$

Second, we incorporate page distance (PD) based on the hypothesis that adding weights to input addresses through the inversion of page distances can improve the model prediction performance:

$$c_{pd}^n = \frac{1}{|page_n - page_1| + 1} \tag{6}$$

where $page_n$ is the page address at history n, $page_1$ is the current page address, the PD input vector is $\mathbf{c_{pd}} = [c_{pd}^1; c_{pd}^2; ...; c_{pd}^N]$.

The context input is the concatenation of $\mathbf{c_{pc}}$ and $\mathbf{c_{pd}}$. A linear projection $\mathbf{E}_{\text{ce}}$ is applied for context embedding that maps the context input vector to the same dimension of the Transformer input. Therefore, the input embedding, position embeddings and the context embedding can be added as described in Equation 7.

$$\mathbf{z}_0' = \mathbf{z}_0 + [\mathbf{c_{pc}}; \mathbf{c_{pd}}]\mathbf{E}_{\text{ce}} \tag{7}$$

3.4.3 Loss Function. For our multi-label classification problem, we use binary cross-entropy loss defined as below:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log\left(p\left(y_i\right)\right) + \left(1 - y_i\right) \log\left(1 - p\left(y_i\right)\right) \tag{8}$$

where y_i is the label and $p(y_i)$ is the predicted probability for sample i being $True$. For multi-label training, each dimension is considered independent and the loss is summed.

3.5 Distance Prefetching

A real hardware implementation will incur some latency. Attention-based model is feasible for high parallel implementations. According to Equation 1 - 3 and Figure 3, the network inference latency under a fully paralleled implementation can be estimated as:

$$\begin{aligned} Latency = & \underbrace{T_{mm_e} + T_{add}}_{\text{Embeddings}} + \underbrace{T_{mm_h} + T_{av}}_{\text{Output head}} + \\ L \times [& \underbrace{4T_{mm_a} + 3T_{av}}_{\text{Multi-head attention}} + T_{mm_f} + 2(T_{add} + T_{norm})] \end{aligned} \tag{9}$$

Transformer layer

where T_{mm_e} is the latency of matrix multiplication for the input embeddings, T_{add} is the vector addition latency, T_{mm_h} is the MLP head latency, T_{av} is the latency for activation functions, mask, and scale operations, T_{mm_a} is the latency of multi-head attention, T_{norm} is the normalization latency. L is the number of Transformer layers.

While attention-based network reduces inference latency, we can further offset the latency by skipping the inference slot and predict the future memory accesses in a distance. The model for distance prefetching can by easily trained through distance labeling, i.e., collecting labels by skipping the estimated inference latency.

4 EXPERIMENTS

4.1 Benchmarks

We evaluate TransFetch and the baselines using the application traces generated from benchmarks *SPEC CPU 2006* [24], *SPEC CPU 2017* [10], and *GAP* [2] using SimPoint [47]. After skipping 1M instructions for warm-up, we use 100M instructions for experiments. We use the first 40M instructions for model training, the next 10M instructions for validation, tuning, and generating optimal thresholds, and the last 50M instructions for evaluation[1].

Table 1: Benchmark statistics

BMKs	# PCs	# Addresses	# Pages	# Deltas
SPEC 06	23~893	60.0K~2.21M	2.51K~88.9K	23.6K~2.01M
SPEC 17	26~1126	62.1K~1.78M	7.99K~ 0.26M	3.18K~0.72M
GAP	63~118	0.56M~1.25M	8.27K~ 27.2K	0.30M~1.20M

Table 1 shows the number of unique program counters (PCs), addresses, page addresses, and deltas. If using tokenization, a token dictionary needs to store the mapping of the unique values to tokens, which is consumes storage. Our method discards tokenization and saves up to table of length 2.01M compared with delta inputs, and up to table of length 0.26M compared with page & offset inputs, given the same level of model complexity.

4.2 Prediction Evaluation

We evaluate the prediction performance of the model by comparing the predicted deltas to the labels. With fixed labels, we vary the model backbones (feature extractor) and inputs to understand the advantages of TransFetch.

4.2.1 Implementation. The configuration of TransFetch is shown in Table 2. The models are trained using ADAM [31] optimizer with decayed learning rate. We set the delta bound as ±128 that can skip the page boundary of 64, which determines the bitmap size as 256.

[1]The code is available at: https://github.com/pgroupATusc/TransFetch.git

Table 2: Model configuration

	Configuration	Value	Configuration	Value
Input/output	Delta bound	±128	Delta bitmap B	256
	History N	9	Look-forward W	128
Attention	Dimension D	128	MLP head layer	1
	Head number	4	Layer L	2

4.2.2 Backbones. To evaluate the contribution of attention layers, under the same input and output configuration in Table 2, we implement three neural networks as the backbones of our framework:

- **LSTM** [18] with hidden dimension = 256, number of layers = 1, and output dimension = 256, indicated as (256, 1, 256).
- **TCN** [33] with hidden dimension same as input sequence length l_{in}, channel = 1, filter size = 4, and output dimension = 256, indicated as (l_{in}, 1, 4, 256).
- **Attention** [66] as in Table 2, with hidden dimension = 128, number of heads = 4, depth = 2, and output dimension = 256, indicated as (128, 4, 2, 256).

4.2.3 Inputs. To evaluate the contribution of address segmentation, we implement three input methods:

- **Delta** input uses the jumps between consecutive memory access addresses. A value-to-token dictionary is required; this requires extra storage space.
- **Page & offset** splits an address only to page address and page offset. The page addresses also need tokenization and use extra storage for token dictionary. The offsets can be directly fed into the model.
- **Address segmentation (AS)** splits an address to segments and avoids tokenization. Particularly, when the segmentation bit = 6, the segment size is same as the block index.

4.2.4 Metrics. We use precision, recall, and F1-score [48] to evaluate the memory access prediction performance of the models.

4.2.5 Output threshold. For all the implemented models, we determine the optimal threshold through a grid search to achieve the highest F1-score in validation.

4.2.6 Results. Table 3 shows the prediction performance of the implemented backbones under various inputs. For a fair comparison, the models are tuned with the same order of complexity, except the TCN whose model size is influenced by the input format. For each backbone, the trend is clear that 1-bit AS and 6-bit AS as inputs result in higher performance than delta input, page & offset inputs, and other AS methods. Specifically, LSTM is stronger in bit-wise input while attention shows higher performance in 6-bit segmentation. Address segmentation achieves 0.26 - 0.36 higher F1-score than delta inputs and 0.15 - 0.24 higher F1-score than page & offset inputs. Comparing among different backbones, we observe that attention-based models typically acquire higher recall than LSTM and TCN, which leads to 0.10 - 0.13 higher F1-score.

4.2.7 Effectiveness of Context Enhancement. To evaluate the influence of context enhancement (CE), we conduct ablation studies on program counter (PC) and page distance (PD). When we introduce both PC and PD, the enhanced model achieves the highest precision and F1-score for all the three benchmarks. The F1-score improves by 3.1% - 9.1% when context enhancement is introduced.

4.3 Prefetching Evaluation

We evaluate the prefetching performance of TransFetch by comparing with the state-of-the-art rule-based prefetchers and ML-based prefetchers. Both the input methods and the labeling methods can be different among these prefetchers. We follow their original designs and compare the overall contributions to the improvement of IPC under the same data set and simulation environment.

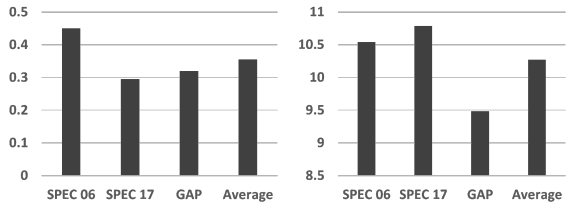
4.3.1 Simulator. We evaluate our approach using the simulation framework released by the 2021 ML-Based Data Prefetching Competition, which is based on ChampSim [5]. The simulator parameters are shown in Table 5. We simulate all prefetchers at the last-level cache (LLC). There is no prefetching for other cache levels.

4.3.2 Baseline Prefetchers. We compare TransFetch with state-of-the-art rule-based prefetchers and ML-based prefetchers:

- **Rule-based prefetchers** including spatial prefetchers BOP [37], VLDP [55], and SPP [30] with variable degree; temporal prefetchers ISB [23] and Domino [1].
- **ML-based prefetchers** including Embedding-LSTM [15] with delta inputs, Clustering-LSTM [15] with address space clustering, and Voyager [56].

4.3.3 Metrics. We use prefetch accuracy, coverage, and IPC improvement [60] to evaluate the prefetching performance. Defining a useful prefetch as the prefetched line being referenced by the application before it is replaced, we have:

- **Prefetch accuracy** as the ratio of useful prefetches to the overall prefetches;
- **Prefetch coverage** as the ratio of useful prefetches to the overall cache misses;
- **IPC improvement** as the percentage increase of instructions per cycle.



(a) Optimal threshold **(b) Mean prefetch degree**

Figure 4: Optimal threshold and mean prefetch degree.

4.3.4 Prefetch Degrees. Figure 4a shows the adapted optimal thresholds for TransFetch. The average thresholds are 0.448, 0.296, and 0.323 for benchmarks SPEC 06, SPEC 17, and GAP, respectively. The overall average threshold is 0.355. Figure 4b shows the mean prefetch degrees after throttled from optimal thresholds. The overall average degree is 10.271 across all applications. According to

Table 3: Comparison of model backbones and input methods

Backbone	Input	# Params (K)	SPEC 06			SPEC 17			GAP		
			Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
LSTM (256,1,256)	Delta*	477	0.5877	0.4218	0.4911	0.6017	0.4215	0.4957	0.5618	0.0670	0.1197
	Page & offset*	625	0.6323	0.4356	0.5158	0.6208	0.4257	0.5051	0.5994	0.1372	0.2233
	1-bit AS	812	**0.7158**	**0.5039**	**0.5914**	**0.6763**	**0.4519**	**0.5418**	0.5835	**0.3594**	**0.4448**
	4-bit AS	451	0.6122	0.4712	0.5325	0.6177	0.4364	0.5115	0.4281	0.3361	0.3765
	6-bit AS†	394	0.6291	0.4520	0.5260	0.6172	0.4383	0.5126	**0.6096**	0.3403	0.4368
	8-bit AS	394	0.6513	0.4938	0.5617	0.6160	0.4457	0.5172	0.5762	0.3311	0.4206
	12-bit AS	369	0.6110	0.4510	0.5190	0.6072	0.4397	0.5101	0.5106	0.3467	0.4130
	16-bit AS	361	0.6022	0.4147	0.4911	0.6519	0.4407	0.5259	0.5406	0.3393	0.4169
TCN (l_{in},1,4,256)	Delta*	889	0.5633	0.4179	0.4798	0.5833	0.4101	0.4816	0.3618	0.1350	0.1966
	Page & offset*	1679	0.6871	0.4921	0.5735	0.6110	0.4433	0.5138	0.5219	0.2024	0.2917
	1-bit AS	8153	0.6992	**0.5298**	**0.6028**	0.6231	0.4833	0.5444	0.5639	0.3451	0.4282
	4-bit AS	436	0.7027	0.5173	0.5959	**0.6679**	0.4623	**0.5464**	0.4522	0.3530	0.3965
	6-bit AS†	201	**0.7135**	0.5134	0.5971	0.6169	**0.4892**	0.5457	**0.5877**	**0.3551**	**0.4358**
	8-bit AS	132	0.6033	0.4138	0.4909	0.6265	0.4335	0.5124	0.4425	0.3223	0.3730
	12-bit AS	43	0.6005	0.4127	0.4892	0.5989	0.4328	0.5024	0.3976	0.3213	0.3554
	16-bit AS	29	0.5908	0.4079	0.4826	0.5972	0.4316	0.5011	0.3966	0.3250	0.3572
Attention (128,4,2,256)	Delta*	431	0.5112	0.4338	0.4693	0.5623	0.3947	0.4638	0.1956	0.1986	0.1971
	Page & offset*	431	0.5989	0.5637	0.5808	0.5699	0.5123	0.5396	0.2676	0.4012	0.3211
	1-bit AS	433	0.6856	0.7439	0.7136	**0.6025**	0.6484	0.6246	0.4020	0.8418	0.5442
	4-bit AS	433	0.6637	0.7337	0.6969	0.6020	0.6118	0.6069	0.3876	0.7337	0.5072
	6-bit AS†	432	**0.6997**	**0.7565**	**0.7270**	0.6018	**0.6901**	**0.6429**	**0.4170**	**0.8429**	**0.5580**
	8-bit AS	432	0.6869	0.6918	0.6893	0.5331	0.5701	0.5510	0.3174	0.7068	0.4381
	12-bit AS	432	0.6799	0.6865	0.6832	0.5796	0.5443	0.5614	0.3431	0.7162	0.4639
	16-bit AS	432	0.6693	0.6935	0.6812	0.5424	0.5215	0.5317	0.3097	0.7020	0.4298

∗ Delta input and page & offset input need tokenization and require extra storage for token dictionaries.
† 6-bit AS is when the segment size equals the block index size, i.e. $s = c$.

Table 4: Ablation study of program counter (PC) and page distance (PD)

Method	PC	PD	SPEC 06			SPEC 17			GAP		
			Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
Basic			0.6997	0.7565	0.7270	0.6018	0.6901	0.6429	0.4170	**0.8429**	0.5580
$+\mathbf{c_{pc}}\mathbf{E}_{ce}$	✓		0.7430	**0.7777**	0.7599	0.8080	0.5314	0.6412	0.6245	0.6268	0.6256
$+\mathbf{c_{pd}}\mathbf{E}_{ce}$		✓	0.8493	0.7117	0.7744	0.7556	0.5881	0.6614	0.5856	0.6377	0.6105
$+[\mathbf{c_{pc}};\mathbf{c_{pd}}]\mathbf{E}_{ce}$	✓	✓	**0.8638**	0.7217	**0.7864**	**0.8144**	**0.6496**	**0.6735**	**0.6634**	0.6344	**0.6486**

Table 5: Simulation parameters

Parameter	Value
CPU	4 GHz, 4-wide OoO, 256-entry ROB, 64-entry LSQ
L1 I-cache	64 KB, 8-way, 8-entry MSHR, 4-cycle
L1 D-cache	64 KB, 12-way, 16-entry MSHR, 5-cycle
L2 Cache	1 MB, 8-way, 32-entry MSHR, 10-cycle
LL Cache	8 MB, 16-way, 64-entry MSHR, 20-cycle
DRAM	$t_{RP} = t_{RCD} = t_{CAS}$ = 12.5 ns, 2 channels, 8 ranks, 8 banks, 32K rows, 8GB/s bandwidth per core

the degree results, we set the overall degree of rule-based baseline prefetchers as 10 for a fair comparison. For ML-based prefetchers, we prefetch predictions with the 10 top probabilities.

4.3.5 Simulation Results. Figure 5 illustrates the prefetch accuracy. Voyager achieves the highest average accuracy at 95.37% among all prefetchers. SPP achieves accuracy at 91.63%, the highest among rule-based prefetchers. TransFetch has accuracy at 88.56% that is the third highest among all prefetchers. VLDP, BOP, ISB, Domino, Embedding-LSTM, and Clustering-LSTM achieve lower accuracy at 85.08%, 75.69%, 46.49%, 22.70%, 70.79%, and 67.54%, respectively.

Figure 6 shows the prefetch coverage. In an average, TransFetch achieves the highest coverage at 60.54%, compared with Voyager at 51.80%. BOP achieves 35.74%, which is the highest among rule-based prefetchers. VLDP, SPP, ISB, Domino, Embedding-LSTM, and Clustering-LSTM achieve lower coverage at 28.09%, 35.47%, 12.80%, 16.57%, 31.86%, and 39.16%, respectively.

Figure 7 shows the IPC improvement of the prefetchers, which indicates the overall contribution of a prefetcher to the system

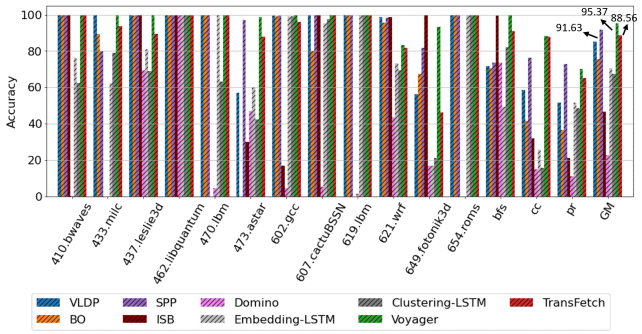


Figure 5: Prefetch accuracy of TransFetch and baselines.

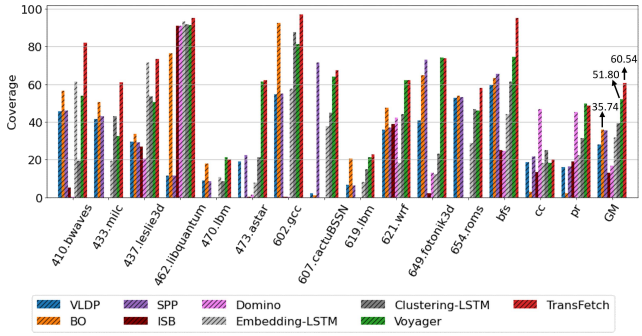


Figure 6: Prefetch coverage of TransFetch and baselines.

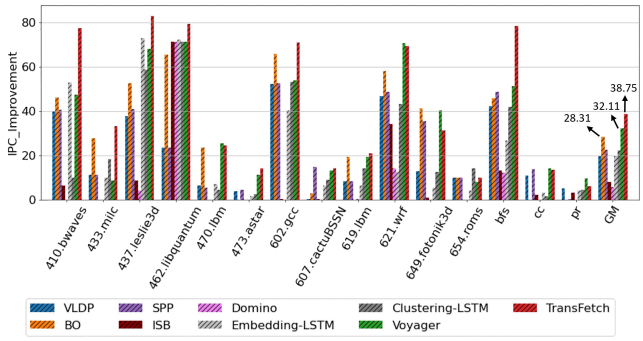


Figure 7: IPC improvement of TransFetch and baselines.

speedup. In the average, TransFetch achieves the highest IPC improvement at 38.75%, which is 10.44% higher than BOP and 6.64% higher than Voyager. VLDP, SPP, ISB, Domino, Embedding-LSTM, and Clustering-LSTM achieve lower IPC improvement at 19.50%, 22.59%, 8.01%, 5.64%, 19.46%, and 21.99%, respectively.

There are cases when TransFetch significantly outperforms other prefetchers. For *410.bwaves*, TransFetch achieves 77.44% IPC improvement, compared with Embedding-LSTM at 52.89% and BOP at 46.12%. For *bfs*, TransFetch ahieves 78.5% IPC improvement, compared with the second highest Voyager at 51.14% and the highest rule-based prefetcher SPP at 48.8%.

4.4 Distance Prefetching Evaluation

In ideal implementation, assuming full parallelism in our model, the estimated latency $T \approx 100$ cycles according to Equation 9 and Table 2, with dimensions $D = 64$, layer L=2, matrix multiplication $T_{mm} = 1+\log_2 D$, and 1 cycle lookup table implemented activations. Recent works have explored more efficient implementations, e.g., replacing matrix multiplication by lookup tables [51] and combinational logic [41]. In future implementations, the range of $T < 200$ can be a reasonable target.

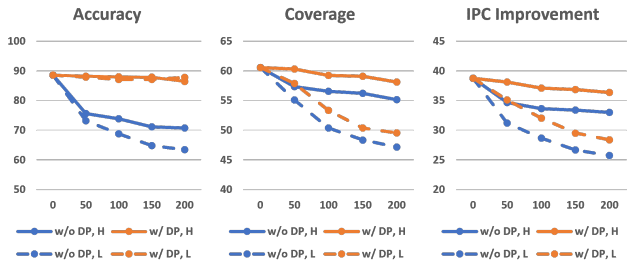


Figure 8: Effectiveness of distance prefetching. "L" means low throughput (1/T); "H" means high throughput (1).

We train and simulate TransFetch with induced latency T from 0 to 200 cycles, under bounds of throughput $1/T$ and 1 inference per cycle. The average prefetching performance is shown in Figure 8. With 200 cycles latency, the high throughput model with distance prefetching (DP) achieves 36.29% IPC improvement, higher than the model without DP at 34.67%, both are still highest compared with the baselines in Figure 7. Even for the low throughput models, DP shows the IPC improvement at 28.39% for 200 cycles latency, slightly higher than BOP at 28.31, while the models without DP drop to 25.78%. Overall, distance prefetching effectively decelerates the performance drop caused by inference latency.

5 CONCLUSION

In this paper, we presented TransFetch, a novel way to model prefetching and to solve the problem of *class explosion, tokenization, labeling*, and *latency*. The keys to our approach are using fine-grained address segmentation for model input to reduce vocabulary and avoid tokenization, using delta bitmap for labeling, and using powerful and parallelizable attention-based network for prediction. TransFetch achieves 26% - 36% higher F1-score than delta inputs and 15%- 24% higher F1-score than page & offset inputs. TransFetch achieves 38.75% IPC improvement in simulation, outperforming the best-performing rule-based prefetcher BOP by 10.44% and ML-based prefetcher Voyager by 6.64%. We believe TransFetch offers a new paradigm for modeling prefetching toward high performance and practicality. In future work, we plan to explore the incorporation of software hints to improve prefetching performance.

ACKNOWLEDGMENTS

This work was supported by National Science Foundation (NSF) under award number CCF-1912680.

REFERENCES

[1] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 131–142.

[2] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[3] Peter Braun and Heiner Litz. 2019. Understanding Memory Access Patterns for Prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*.

[4] Carlos Carvalho. 2002. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*.

[5] "ChampSim". 2017. https://github.com/ChampSim/ChampSim.

[6] Chi F Chen, S-H Yang, Babak Falsafi, and Andreas Moshovos. 2004. Accurate and complexity-effective spatial pattern prediction. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 276–287.

[7] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers* 44, 5 (1995), 609–623.

[8] Trishul M Chilimbi. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. *ACM SIGPLAN Notices* 36, 5 (2001), 191–202.

[9] Yuan Chou. 2007. Low-cost epoch-based correlation prefetching for commercial applications. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 301–313.

[10] "SPEC CPU2017". 2017. The Standard Performance Evaluation Corporation. https://www.spec.org/cpu2017/.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[13] Michel Dubois, Murali Annavaram, and Per Stenström. 2012. *Parallel computer organization and design*. cambridge university press.

[14] Keith I Farkas, Paul Chow, Norman P Jouppi, and Zvonko Vranesic. 1997. Memory-system design considerations for dynamically-scheduled processors. *ACM SIGARCH Computer Architecture News* 25, 2 (1997), 133–143.

[15] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. *arXiv preprint arXiv:1803.02329* (2018).

[16] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. *CoRR* abs/1803.02329 (2018). arXiv:1803.02329 http://arxiv.org/abs/1803.02329

[17] Anakhi Hazarika, Soumyajit Poddar, and Hafizur Rahaman. 2020. Survey on memory management techniques in heterogeneous computing systems. *IET Computers & Digital Techniques* 14, 2 (2020), 47–60.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[19] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. 2003. TCP: Tag correlating prefetchers. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, 317–326.

[20] Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 397–408.

[21] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. 2018. Ai benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 0–0.

[22] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* 13, 2011 (2011), 1–24.

[23] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 247–259.

[24] Aamer Jaleel. 2010. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload* (2010).

[25] Teresa L Johnson, Matthew C Merten, and Wen-Mei W Hwu. 1997. Run-time spatial locality detection and optimization. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 57–64.

[26] Doug Joseph and Dirk Grunwald. 1997. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*. 252–263.

[27] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 364–373.

[28] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[29] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. 2014. Trends in big data analytics. *Journal of parallel and distributed computing* 74, 7 (2014), 2561–2573.

[30] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[31] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[32] Sanjeev Kumar and Christopher Wilkerson. 1998. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. IEEE, 357–368.

[33] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. 2017. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 156–165.

[34] Wei-Fen Lin, Steven K Reinhardt, Doug Burger, and Thomas R Puzak. 2001. Filtering superfluous prefetches using density vectors. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*. IEEE, 124–132.

[35] Ward Douglas Maurer and Ted G Lewis. 1975. Hash table methods. *ACM Computing Surveys (CSUR)* 7, 1 (1975), 5–19.

[36] Julian Richard Medina and Jugal Kalita. 2018. Parallel attention mechanisms in neural machine translation. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 547–552.

[37] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 469–480.

[38] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[39] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. 2011. Natural language processing: an introduction. *Journal of the American Medical Informatics Association* 18, 5 (2011), 544–551.

[40] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*. 48–53.

[41] Mahdi Nazemi, Arash Fayyazi, Amirhossein Esmaili, Atharva Khare, Soheil Nazar Shahsavani, and Massoud Pedram. 2021. NullaNet Tiny: Ultra-low-latency DNN Inference Through Fixed-function Combinational Logic. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 266–267.

[42] Kyle J Nesbit, Ashutosh S Dhodapkar, and James E Smith. 2004. AC/DC: An adaptive data cache prefetcher. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 135–145.

[43] Tesla NVIDIA. 2017. V100 GPU architecture. *The world's most advanced data center GPU. Version WP-08608-001_v1* 1 (2017).

[44] Subbarao Palacharla and Richard E Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*. 24–33.

[45] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 285–297.

[46] Leeor Peled, Uri Weiser, and Yoav Etsion. 2018. A neural network memory prefetcher using semantic locality. *arXiv preprint arXiv:1804.00478* (2018).

[47] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (2003), 318–319.

[48] David MW Powers. 2020. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061* (2020).

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

[50] S Rahman, M Burtscher, Z Zong, and A Qasem. 2015. Maximizing Hardware Prefetch Effectiveness with Machine Learning. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 383–389.

[51] Mohammad Samragh Razlighi, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. 2017. Looknn: Neural network with no multiplication. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1775–1780.

[52] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2019. Survey and benchmarking of machine learning accelerators. In *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE, 1–9.

[53] Takaya Saito and Marc Rehmsmeier. 2015. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PloS one* 10, 3 (2015), e0118432.

[54] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, et al. 2018. Graphchallenge. org: Raising the bar on graph analytic performance. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[55] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 141–152.

[56] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 861–873.

[57] Alan Jay Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer* 11, 12 (1978), 7–21.

[58] Yan Solihin, Jaejin Lee, and Josep Torrellas. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 171–182.

[59] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 252–263.

[60] Viji Srinivasan, Edward S Davidson, and Gary S Tyson. 2004. A prefetch taxonomy. *IEEE Trans. Comput.* 53, 2 (2004), 126–140.

[61] Ajitesh Srivastava, Angelos Lazaris, Benjamin Brooks, Rajgopal Kannan, and Viktor K Prasanna. 2019. Predicting memory accesses: the road to compact ML-driven prefetcher. In *Proceedings of the International Symposium on Memory Systems*. 461–470.

[62] Ajitesh Srivastava, Ta-Yang Wang, Pengmiao Zhang, Cesar Augusto F De Rose, Rajgopal Kannan, and Viktor K Prasanna. 2020. MemMAP: Compact and Generalizable Meta-LSTM Models for Memory Access Prediction. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 57–68.

[63] Chun-Wei Tsai, Chin-Feng Lai, Han-Chieh Chao, and Athanasios V Vasilakos. 2015. Big data analytics: a survey. *Journal of Big data* 2, 1 (2015), 1–32.

[64] Raju Vaishya, Mohd Javaid, Ibrahim Haleem Khan, and Abid Haleem. 2020. Artificial Intelligence (AI) applications for COVID-19 pandemic. *Diabetes & Metabolic Syndrome: Clinical Research & Reviews* 14, 4 (2020), 337–339.

[65] Steven P Vander Wiel and David J Lilja. 1997. When caches aren't enough: Data prefetching techniques. *Computer* 30, 7 (1997), 23–30.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[67] Brian Wahl, Aline Cossy-Gantner, Stefan Germann, and Nina R Schwalbe. 2018. Artificial intelligence (AI) and global health: how can AI contribute to health in resource-poor settings? *BMJ global health* 3, 4 (2018), e000798.

[68] Jonathan J Webster and Chunyu Kit. 1992. Tokenization as the initial phase in NLP. In *COLING 1992 Volume 4: The 14th International Conference on Computational Linguistics*.

[69] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal streams in commercial server applications. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 99–108.

[70] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 996–1008.

[71] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient metadata management for irregular data prefetching. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–13.

[72] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[73] Yuan Zeng and Xiaochen Guo. 2017. Long short term memory based hardware prefetcher: a case study. In *Proceedings of the International Symposium on Memory Systems*. 305–311.

[74] Pengmiao Zhang, Ajitesh Srivastava, Benjamin Brooks, Rajgopal Kannan, and Viktor K Prasanna. 2020. RAOP: Recurrent Neural Network Augmented Offset Prefetcher. In *The International Symposium on Memory Systems (MEMSYS 2020)*.

[75] Pengmiao Zhang, Ajitesh Srivastava, Ta-Yang Wang, Cesar AF De Rose, Rajgopal Kannan, and Viktor K Prasanna. 2021. C-MemMAP: clustering-driven compact, adaptable, and generalizable meta-LSTM models for memory access prediction. *International Journal of Data Science and Analytics* (2021), 1–14.