Drishyam: An Image is Worth a Data Prefetcher

Shubdeep Mohapatra
Dept. of Computer Science and Engineering
BITS Pilani
Goa, India
shubdeepmohapatra01@gmail.com

Biswabandan Panda
Dept. name of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India
biswa@cse.iitb.ac.in

***Abstract*—Hardware prefetching is a latency-hiding technique that hides the costly off-chip DRAM accesses. Although hardware prefetching is an extensively researched topic with many state-of-the-art data prefetchers pushing the performance limits, prefetching for irregular applications with hard-to-predict access patterns is still a challenging problem to solve. The usage of neural networks for hardware prefetching is a promising direction, especially for predicting irregular memory access patterns. This paper presents Drishyam, a novel hardware prefetcher based on computer vision algorithms that use images to learn memory access patterns and predict future memory accesses with high accuracy and coverage. For hardware prefetching, an image is a graphical representation of memory accesses observed over time. For a sequence of memory addresses, Drishyam creates images that predict the future addresses by predicting the future OS page and a cache line offset within the OS page. Drishyam outperforms Voyager, the state-of-the-art machine learning (ML) based prefetcher, for a set of irregular benchmarks by an average of 4.7% with an average prefetch accuracy and prefetch coverage of 89.5% and 66.6%, respectively. In terms of training time, Drishyam outperforms Voyager by 225.5%.**

***Index Terms*—Cache, Prefetching, Performance**

I. Introduction

Hardware data prefetching is a latency-hiding technique that proactively brings data into the caches, effectively converting cache misses into hits. State-of-the-art data prefetchers [1]–[5] have pushed limits of hardware prefetching with 2 to 3% performance improvement over previous state-of-the-art techniques. However, these prefetchers lose effectiveness and perform poorly for irregular benchmarks with hard-to-predict memory access patterns.

Machine learning (ML) can be used for better hardware prefetching [6]–[10], keeping practical aspects aside. A recent ML-based prefetcher called Voyager [10] is a top-performing prefetcher that solves some of the challenges faced by previous ML approaches. One of the fundamental problems while applying ML techniques on data prefetching is the *class explosion problem*. For a 64-bit address space, access to address X can be followed by a 64B cache-line aligned delta (any address in the cache-line aligned 58-bit address space), which is huge (2^{58}). So, applications of ML techniques on absolute addresses (as classes) are not a feasible solution. However, the class explosion problem can be solved by breaking the prediction of the delta problem into two sub-problems: (i) OS page prediction and (ii) offset prediction within a given page [10].

Why computer vision for prefetching? Computer vision requires images to analyze and discern distinctions and patterns between them to recognize the images. The usage of images helps in capturing the temporal correlations among memory accesses. We are motivated by the wild and crazy idea (WACI) that appeared in ASPLOS 2022, WACI session [11]. While the number of memory accesses for a program can range in millions, using images reduces that size, decreasing the training and inference time. For example, generating an image with ten consecutive memory accesses reduces a data set consisting of a hundred memory accesses into a data set of ten images.

We propose Drishyam[1], a prefetcher that uses images to generate prefetch requests. Drishyam creates two images for a memory address; one for a memory region (like a 4KB OS page) and one for the offset within the page. For a 4KB page with a cache-line size of 64 bytes, there are 64 possible cache-line offsets. Instead of using the page numbers and offsets, the images for pages are created using their embeddings and for offsets using their embeddings, which also contain the context of the page. Drishyam feeds these images to a transformer [12] to predict the future addresses that should be prefetched.

Our approach. For prefetching, we create two sets of images, one for the page accessed and another for the corresponding offset in that page which was referred. This division of tasks makes prediction easier as the number of unique addresses accessed in a program can range in millions. Still, the number of pages is only in the range of thousands (minimum 4K with `pr` and maximum 90K with `mcf`), whereas the number of offsets is fixed at 64.

Figure 1 shows an image (16 rows and 16 columns) corresponding to 16 memory accesses that can get mapped to 16 different pages. Each of these rows corresponds to one of the pages. The goal of this image creation is to predict the future page. Similar images can be created for predicting the offsets. To create meaningful images, we need to represent the pages and offsets as numerical values. That is where *embeddings* come in and represent non-numerical data into low-dimensional numerical vectors. An embedding is an internal representation of input features within a neural network.

A major part of the work was done through a summer internship, while the author was at IIT Bombay.

[1]Drishyam is a `Sanskrit` word meaning visual.

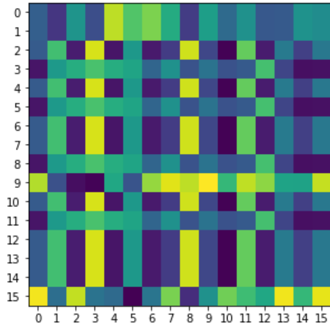

Fig. 1. An image representing memory accesses corresponding to 16 pages for the `mcf` benchmark.

An embedding layer learns the corresponding representation such that input features that behave similarly have similar embeddings. In Figure 1, the columns, therefore, represent the features that help in distinguishing the two pages. In this case, the embedding vector for a page is a 16-dimensional vector, where the weight of each component is in the range [0,1]. Figure 1 is a YBG spectrum image meaning if the weight is 1, then the pixel color is yellow; green if the weight is 0.5, and blue if the weight is 0. It means that if the embedding component is close to zero, the pixel color is blue, and if it is close to one, the color is in the yellow spectrum. So, in a nutshell, rows in Figure 1 represent the embedding vectors for individual pages. For example, rows 12, 13, and 14 have the same color distribution for each column. This implies that they are visual representations of the same pages. Drishyam creates these vectors so that similar inputs will have similar embeddings.

Based on Figure 1, we create images wherein one image contains the information about 16 recent memory accesses, and k images represent $k \times 16$ memory accesses. We assign a class to each image, such that the class associated with it will be the memory address(more specifically the page and offset associated with the address) that will be accessed after 16 accesses that are used to create the image. Therefore our prediction process boils down to a classification process where we predict the class, to which an image belongs. The classification is performed by the SWIN transformer [12].

To the best of our knowledge, this is the first work that makes a case for the usage of images for accurate data prefetching. Note that this paper does not make a case practical use of computer vision algorithms for use in hardware data prefetchers.

Overall, our contributions are as follows:

- We make a case for a hardware prefetcher that uses images and image classification algorithms to learn future memory access patterns (Sections III and IV).
- We use a SWIN transformer with self attention across shifted windows to improve our prediction accuracy (Section IV-C).
- We explain the design of Drishyam (Sections V and VI). Compared to the state-of-the-art ML prefetcher Voyager, Drishyam provides an average 4.7% performance boost with an average prefetch accuracy and coverage of 89.5% and 66.6%, respectively (Section VII). Compared to Voyager, Drishyam is 225% faster in terms of training time.

II. Recent advances in ML for data prefetching

Hashemi et al. [9] formulate data prefetching as a classification problem and use Long Short-term Memory (LSTM) [13] for prefetching. However, their proposal can only learn access patterns within a spatial region. Voyager [10], on the other hand, can perform prefetching that crosses the spatial boundaries and is the high-performing ML prefetcher. The 1st ML for prefetching championship co-located with ISCA 2021 [14] has some interesting ML prefetchers. However, none of them perform better than Voyager, and in some cases, perform worse than the baseline system with no prefetching. Usage of reinforcement learning (RL) for prefetching [6] [15] is also a promising approach. However, the table-based RL methods are inefficient and do not provide significant performance improvement for irregular benchmarks.

III. Data Prefetching and Image Classification

Image classification is a procedure of mapping pixel features to classes. The classifier receives a pixel feature ***x*** and maps it to any one of the ***n*** classes $C_1, C_2, ..., C_n$:-

$$f(x):\ x \rightarrow \Delta\ ; x \in \mathrm{R}^m,\ \Delta = \mathrm{C}_1, C_2, ..., C_n$$

where the dimension of the pixel vector is m, and the number of classes is n. f is a function that assigns the pixel vector x to a single class in the set Δ.

The role of an image classifier is to distinguish one class from others; therefore it performs as a discriminant. This is done with the help of a discriminant function. The discriminant function takes as input the pixel feature x and a class C_k. If a particular pixel feature belongs to a class x, then the discriminant value for that class would be the highest:-

$$g(x,\mathrm{C}_y) > \mathrm{g}(\mathrm{x},C_z),\ \mathrm{z=1,....,n;\ z} \neq y; x \in \mathrm{R}^m$$

Here g is the discriminant function that correlates the pixel feature vector with the class it belongs in.

Problem formulation. A robust problem formulation is essential for any machine-learning problem. For our case, we use probability distribution to model data prefetching as an ML problem. The output is in the form of a probability distribution. We use the unique pages and offsets that an application encounters as classes. Consider a sequence of 17 memory accesses to 17 different pages; the first 16 accesses are used to create an image. The 17th access is the class associated with that image. This procedure is carried out for an entire application. Finally, we get a labeled dataset for the prediction process. The upper limit for the number of classes is the number of unique pages encountered in the application. Creating the dataset for offset prediction has the same process, but the number of classes is fixed at 64 since there are only 64 possible offsets within a page size of 4KB. The output of the classifier, therefore, is the probability of an image belonging to a class given the features. Figure 2 explains this in detail.

Figure 1 illustrates how classes are assigned to an image.

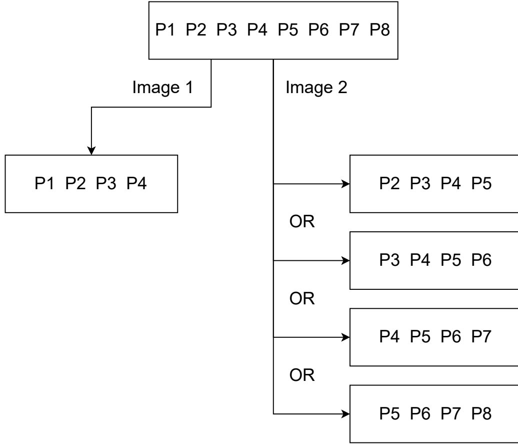

Fig. 2. Images with different widths created with an image length of four.

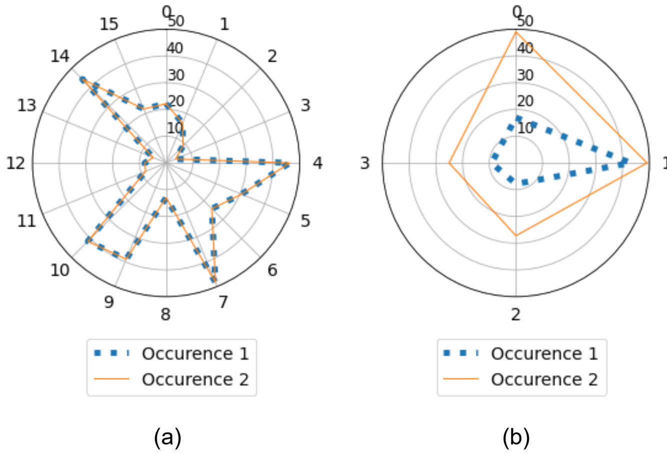

Fig. 3. Radar chart of offset embeddings (a) without page context (b) with page context chart for the two random occurrences of the offset nine in `astar` benchmark. The Y axis denotes the magnitude of the individual components of the vector.

The image in Figure 1 is generated from a sequence of 16 accesses that go to 16 pages (not necessarily unique). The goal of using image classification is that given an image generated from a sequence of accesses $A_1, A_2, ...A_n$, we have to provide the probability that a future address *Addr* will be accessed. *Addr* can thus be seen as a class the image may belong to. Consider an image ***I*** generated from accesses $A_1, A_2, ...A_n$, the learning task, therefore, becomes $P(Addr|x)$, where x is the pixel vector extracted from the image I. In ML parlance, x is the *input feature*, and *Addr* is the *output label*.

IV. Generating Images for Predictions

Image length is the number of consecutive memory accesses (page and offset) used to create an image.

Image width is the frequency at which images are created,i.e. every one access or every two accesses, and so on. For example, If an image-I is created for accesses `a1,a2,a3,...,a16`, then width of one means image-II is created for accesses `a2,a3,...,a17` and width two means image-II is created from accesses `a3,...,a18`.

How frequently do we create images? Figure 2 shows four different sets of images created from eight consecutive accesses that may go to eight different pages. There are multiple options for an image of length four that tracks four pages. Image-I can be created from page 1 to page 4 (P1 to P4). Image-II can have pages P2 to P5, P3 to P6, P4 to P7, or P5 to P8, based on the image *width*.

What resolution should the images be? We use the image resolution 16X16 to visualize page and offset accesses. However, larger resolutions can be used. The trade-off in deciding the resolution is how dense an image will become and whether it will cause information loss that can affect the prediction accuracy. For the irregular benchmarks that we study, we find a resolution of 16×16 is the best, and the effectiveness goes down slightly when we move to 32×32 or 64×64, and it becomes worse beyond 64×64.

Color images vs. grey-scale images. If we use color images, the processing time will be longer but since we can use more bands of color, we can capture more information. For grey-scale images, the processing time will be less, but we will lose relevant information. In general, color images convey more information. Grey images do well for regular benchmarks like `gcc`, but with irregular benchmarks like `mcf`, we find a drop of 32% in prediction accuracy.

A. How to create images from memory accesses?

As a program can access millions of unique addresses, it is easier to divide the prediction task into two stages:-page predictions and offset predictions. The procedure for creating images for both predictions is slightly different. In offset prediction, there is an added complication of distinguishing the same offsets from different pages. This problem is referred to as the *offset aliasing* problem. Consider an offset ***O*** and two pages P_1 and P_2. If the offset from the two pages is not differentiated, then the pixel intensities of the two occurrences of the offset would be the same. As a result, the classifier sees the same input for offset residing on different pages. This would force the classifier to learn an average behavior and lead to inaccurate predictions.

Figure 3(a) is a radar chart of offset embeddings without the page context for two random occurrences of offset nine for the `astar` benchmark. It shows us that without the context of the page the accesses reside in, the classifier fails to differentiate between offsets residing on different pages. Figure 3(b), on the other hand, shows the offset embeddings with page context for the same occurrences of offset nine as in figure 3(a). Adding the page context distinguishes the offset embeddings, and the classifier can now see the same offsets residing on different pages, separately. While calculating the attention for a vector v1 of dimension M with respect to another vector v2 of dimension N, such that N>M, the number of "experts" assigned to calculate the attention vector is N/M. The resulting attention vector for v1 has a dimension of N. In Figure 3(a), the offset embedding vector has a dimension of 16. In Figure 3(b), upon calculating the revised offset embedding by calculating

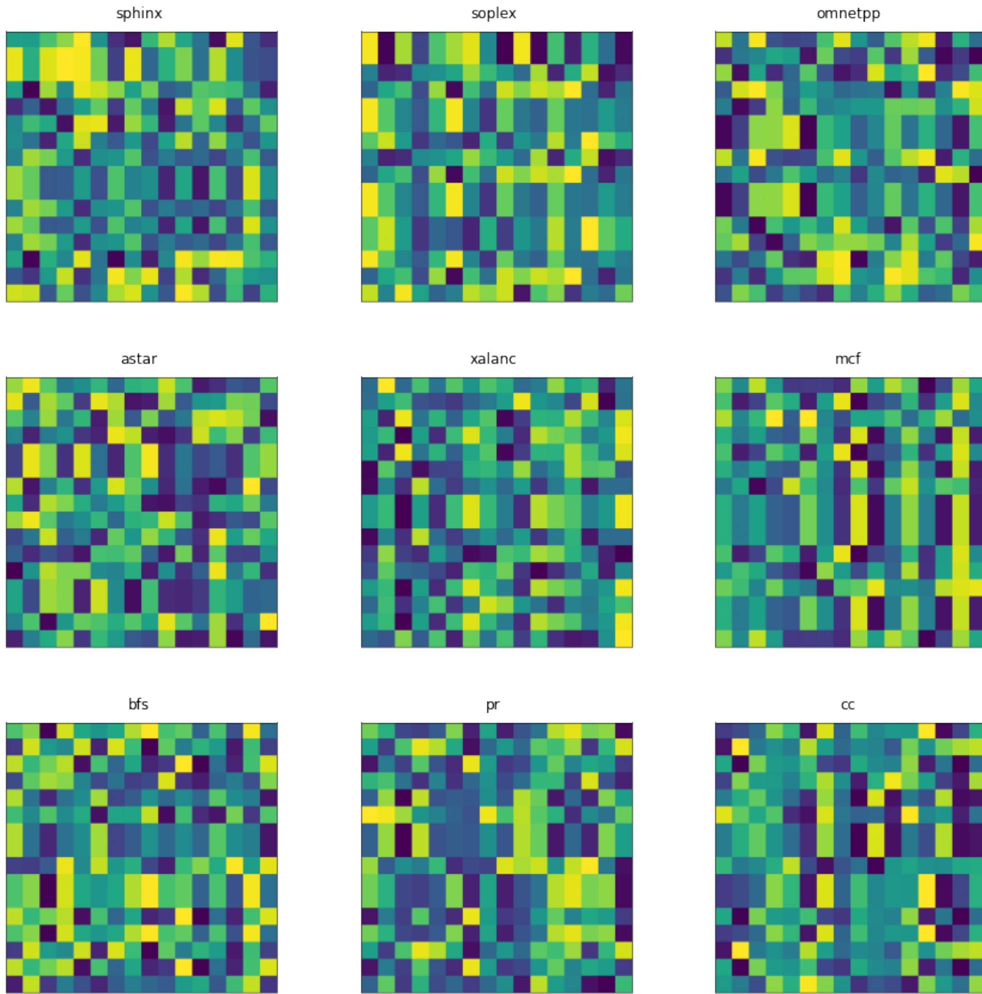

Fig. 4. Representative images for the irregular benchmarks.

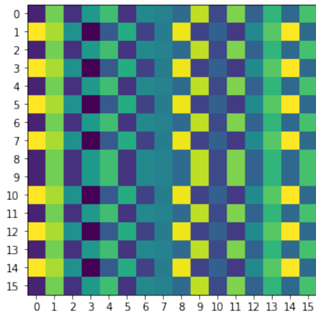

Fig. 5. An image depicting regular access pattern in `gcc` benchmark.

the attention with respect to a page embedding vector of dimension four, we get the output vector with dimension four.

The embedding layer is trained in a way so that the features in terms of pages and offsets closer to each other will have similar embeddings. In simple terms, if two pages are next to each other then the cartesian distance between their embeddings would be lesser than the distance between a page further than them. In layman's terms, if we were to generate simple embeddings for "foot" and "football", their embeddings would be similar. However, if we train the embedding layer to distinguish between body part and sport then the embeddings for both would be different. Figure 4 shows representative images of page accesses for various irregular benchmarks. Figure 5 shows an image of a regular benchmark `gcc` where consecutive accesses are spread across two pages in contrast to many pages with page jumps in Figure 4. The pages accessed for the images are as follows:-

i)sphinx:-41059520835, 10980916361, 10980916361, 68179700096, 41059520846, 41059520847, 41059520821, 41059520850, 41059520851, 41059520851, 41059520850, 41059520851, 41059520850, 41059520824, 41059520838, 41059520736

ii)soplex:-30557552214, 30557552214, 30557552215, 43398299683, 43398299684, 43398299684, 43398299690, 30557552215, 43398299683, 43398299684, 43398299684, 43398299690, 43398299690, 43398299684, 43398299683, 30557552209

iii)omnetpp:-2719162767, 37265539527,

37265539528, 37265539529, 37265539531, 10980916358, 10980916358, 2719162767, 37265539520, 10980916358, 10980916358, 2719162766, 2719162761, 37265539536, 37265539539, 37265539540
iv)astar:-10980916364, 2719162762, 2719162763, 41059520725, 41059520726, 41059520726, 41059520727, 41059520728, 10980916364, 41059520729, 41059520730, 41059520731, 41059520732, 41059520732, 41059520733, 41059520734
v)xalanc:-10980916365, 10980916366, 10980916368, 10980916361, 10980916368, 10980916369, 10980916370, 10980916361, 10980916361, 10980916370, 68179699905, 68179699906, 10980916361, 68179699909, 10980916361, 68179699911
vi)mcf:-43398299714, 29107545297, 29107545295, 63938738763, 29107545297, 63938738763, 63938738665, 63938738763, 63938738763, 63938738763, 29107545297, 29107545467, 29107545301, 63938738763, 63938738763, 43398299714
vii)bfs:-10980916362, 10980916361, 10980916370, 10980916371, 10980916374, 10980916364, 10980916374, 10980916374, 10980916367, 10980916362, 10980916362, 10980916370, 10980916362, 10980916361, 10980916370, 10980916365
viii)pr:-10980916362, 10980916361, 10980916370, 10980916371, 10980916374, 10980916364, 10980916374, 10980916374, 10980916367, 10980916362, 10980916362, 10980916370, 10980916362, 10980916361, 10980916370, 10980916365
ix)cc:-10980916362, 10980916361, 10980916369, 10980916370, 10980916373, 10980916364, 10980916373, 10980916373, 10980916360, 10980916362, 10980916362, 10980916369, 10980916362, 10980916361, 10980916369, 10980916365

B. Adding Page context to offset embedding

To differentiate the same offset on different pages, we add the page context to the offset embedding. This, in simple terms, means that we add the information of the page, on which the offset resides. We do this with the help of an *attention layer*. Attention is commonly used in Natural Language Processing(NLP), where it is used to put more emphasis on certain words. The attention layer takes as input a *query* and a set of *keys* and *values*. It then captures the correlation between the key and queries and provides as output a weighted sum of the values. The weights used are the correlations captured between the key and queries. Consider the dimension of the offset embedding vector to be *n* and the page embedding vector to be *m*. The page embedding vector will be the *query*, and the offset embedding vector will be divided into n/m smaller vectors. These smaller vectors will be the *keys*. The attention layer calculates the correlation of the query with the keys and outputs a probability vector. This probability vector is of dimension n/m and captures the correlation of each key with the query. Finally, we use the offset embedding as *values* to calculate the weighted sum. Consider a page vector *p* and the offset vector o. The correlation between page and offset is captured through:- $W_t = softmax(p.o_t); 0 \leq t < n/m$ where, $softmax(x_i) = \frac{\exp x_i}{\sum_{i=1}^{k} \exp x_i}$ We use dot product attention to capture the correlation between the keys and the query. These weights are then used to get the revised offset embedding $o^{'}$.

$$o^{'} = \sum_{t=1}^{s} W_t o_t; s = n/m$$

C. SWIN Transformers and Prefetching

At a high level, a transformer takes as input an input sequence and converts it into an encoding vector, and decodes it back into another sequence. At the heart of this procedure is the attention mechanism. The other important constituents of the transformer are the encoder and the decoders. The encoders convert their input into another sequence of vectors called encodings and decoders do the reverse. VIT Transformer(Vision Transformer) [16] could work well for image classification tasks. However, there were challenges in using VIT for tasks that require pixel-level prediction, like object segmentation [17] or object detection [18]. SWIN transformer stands for *Swin WINdow* transformer. SWIN transformers are computationally more effective than regular VITs. For VIT transformers, if there are n patches, the attention for a patch is computed with the remaining $n-1$ patches. So the total operations will be O(n^2), whereas, for SWIN, we only compute attention for patches within a window. SWIN transformer has four major parts of the architecture:-patch Partition, patch merging, embedding layer, and the SWIN transformer block. For astar, we show how SWIN architecture works with a representative image (Figure 6).

Patch partition. An image passed through the SWIN transformer is divided into non-overlapping patches. We pass a 16×16 image through the transformer and divide it into patches of size 2×2, in total 64 patches (Figure 6).

Patch Merging. Patch Merging is important for the SWIN transformer as it helps the model make some assumptions on input data it has not seen yet. This is more commonly known as *inductive bias.* This introduction of inductive bias contributes to the improvement of SWIN transformers more than other vision-based transformers. Patch merging combines windows and merges them into one window. This downsamples the feature map size by 2X and increases the depth of each patch by 2X (Figure 6).

Embedding layer. The next component in the SWIN architecture is the linear embedding layer. This layer calculates the patch embedding for each patch and their positional embedding. The usage of position information makes the model more robust. The final output is the addition of these

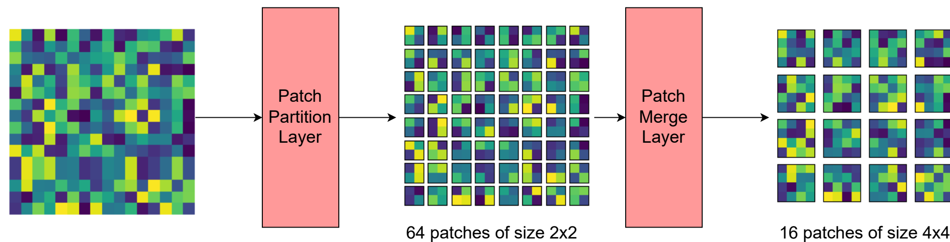

Fig. 6. Visual representation of the patch partition and merge layer.

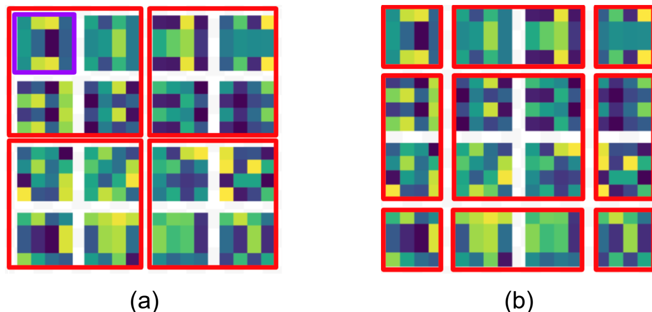

Fig. 7. (a) W-MSA: attention for the top left patch(purple border) is calculated by just taking the dot product with the remaining patches in its window (red border) (b) SW-MSA: windows in the previous image have now been shifted, and patches which were in separate windows previously were now in the same window. Attention for a patch is now calculated by taking the dot product with the patches in the new window.

two embeddings.

Attention mechanism. Before moving on to the SWIN transformer block, we describe the changes made in the attention mechanism. Both SWIN and VIT use encoder blocks adopted from the original transformer architecture. The encoders comprise a multi-headed self-attention layer and a feed-forward network. However, VIT's multi-headed self-attention uses dot product attention to compute the attention of one patch w.r.t. all other patches in the image. This means that while calculating the attention for one patch, we take the dot product with all the other patches. However, with SWIN, we only need to take dot products of the patches within a window. Consider Figure 7(a); if we want to calculate attention for the top left patch, then we have to take the dot product with all other patches. This is the source of the quadratic time complexity for VIT. SWIN transformers resolve this by using custom attention methods: namely, Multi-head Self Attention in a Window (W-MSA) and Multi-head Self Attention Across "Shifted" Windows (SW-MSA).

Multi-head Self Attention in a Window (W-MSA) In SWIN, to compute attention, it takes a fixed-size window. In our case, Figure 7(a), the window dimension is $2{\times}2$, i.e., it consists of 4 patches. To compute attention for the top left corner patch, we only have to attend to patches in the same window. This is much more computationally effective and scalable than attending to all patches.

Multi-head Self attention across "shifted" windows (SW-MSA) A correlation between windows is also important for computer vision-related tasks. SW-MSA is introduced. Figure 7 visually represents its functioning. The windows represented in Fig 7(a) have been shifted, and new windows are created. In the new windows created (Figure 7(b)), patches that were in separate windows before now lie in the same window. Attention is calculated within the new windows, and this ensures that we get the relation between patches that were previously in another window. In summary, to capture the attention between windows, we (i) shift the output of W-MSA by half their height and width. (in our case by 1) and (ii) compute W-MSA in the shifted windows.

SWIN transformer block. The SWIN Transformer block is the backbone of our prefetcher. It consists of two encoders placed in series. We feed the output of the first one to the second encoder. The first encoder computes W-MSA, i.e. attention within windows. The second encoder computes SW-MSA, i.e. attention between windows. The major change in the transformer block is that instead of a simple global multi-headed self-attention, SWIN uses W-MSA and SW-MSA.

Why SWIN transformer for prefetching? One thing to note is that when the transformer is fed patches of the images, it looks at a part of the 64-bit address instead of the entire 64-bit address as the patches are formed over the 2D image and each row is a data point of 1D. If we consider an entire row as a signal, then the transformer is fed a part of the signal. In a conventional transformer, the entire signal is given as input but for irregular benchmarks, since finding an access pattern between the inputs is difficult, the relationship between components of signals can be exploited. Consider a signal sub-sequence S1 and S2 such that whenever these two occur page P1 is accessed. It is difficult to find this pattern when considering the entire signals as the signals may vary but if they contain this sub-sequence then the pattern can be learned.

V. Drishyam Prefetcher

This section describes our computer vision model for data prefetching. Our solution showcases that images can be a new avenue for data prefetching. We first summarise our approach into three key steps in our prediction process:

Efficient representations for memory addresses. We divide the memory accesses into pages and offsets for a smaller output space, making the prediction process easier. Since the individual pages and offset have no numerical meaning, we convert them into a suitable representation.

Creating images for the memory accesses. After getting the alternate representation for the pages and offsets, we create images using those representations. As the alternate

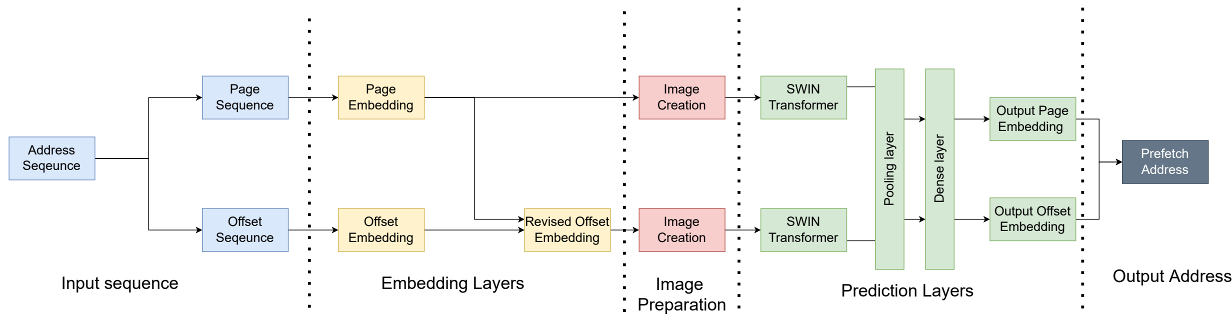

Fig. 8. Drishyam: Neural Architecture

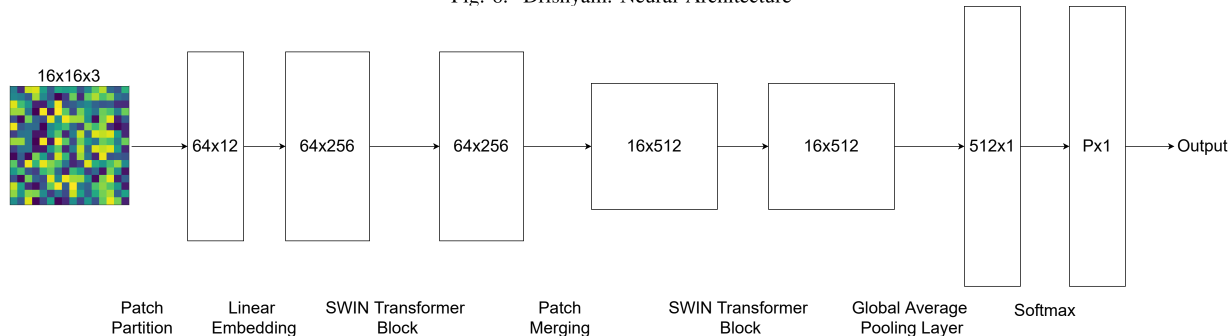

Fig. 9. Different layers of the SWIN transformer in action for Drishyam's prediction process.

representations of the pages and offsets are of dimension H, we create images of dimension H×H (image length is H).

Prediction using SWIN transformer. The images created are then fed to the SWIN transformer. The output is a one-hot encoding vector for pages and offsets.

Figure 8 shows the neural architecture for our prefetcher. The input to the prefetcher is a sequence of addresses of length H; broken down into pages and offsets. The first section of the prefetcher is the embedding layers. As mentioned before, the function of these layers is to convert the page and offsets into real numbers such that inputs that behave similarly have similar embeddings. We calculate the embeddings for the page and offsets separately in the first embedding layer. The embedding dimension for the pages is H, therefore, the output from the page embedding layer is of shape H×H. The output from the first offset embedding layer is of shape H×N, where $N > H$. The embedding dimension is greater here as we are yet to add the page context to this offset embedding. The revised offset embedding is calculated using the *attention layer*. The output from this layer is of shape H×H. Now that we have embeddings for both pages and offset ready, we form images using these embeddings. The images are of dimension H×H, where each pixel represents one component of the embeddings. The next layer takes these images as input and uses two separate SWIN transformers for each page and offset images. The output from the SWIN transformer is fed into a pooling layer for feature summarization, fed into a dense layer with a *softmax* activation function. The output from the dense layer is a probability distribution over the possible pages and offsets. The page and offset with the highest probability are chosen separately, and the prefetch address is generated by adding the offset within a page number. Table I shows

TABLE I
HYPERPARAMETERS FOR TRAINING THE PREFETCHER.

Sequence Length(i.e. Image Length)	16
Image width	8
Embedding dimension for pages (H)	16
Embedding dimension for offsets (N)	64
Frequency of page Number of heads	4
Image dimension	16x16
SWIN transformer hper-parameters	
Patch, window, and shift size	2, 2, and 1
Learning rate	0.001
Batch Size	128
Dropout Rate	0.03
Optimizer	Adam [19]

hyperparameters used for training Drishyam.

Learning rate is a hyperparameter that governs the rate or speed at which the neural network learns the value of the parameter estimates. In simpler terms, it indicates how often the network changes its knowledge. A lower learning rate means less number of changes.

Batch size. Since the amount of data that has to be handled by a neural network can range from tens of thousands to millions; it will be impossible for the neural network to learn from the entire dataset at the same time. Batch size denotes the number of data points currently processed by the network. The weights of the neural network are updated after each batch is processed. Batch size has to be selected carefully as a small batch size can lead to over-fitting.

Dropout rate. Neural networks suffer from a problem called co-adaptation, where multiple neurons extract the same feature. This could lead to overfitting if the duplicate features are characteristic of the training set. To overcome this issue, some neurons of the network are randomly shut down(values are zeroed). The fraction of neurons to be shut down is the

dropout rate.

Optimizer algorithms change the values of the weights and learning rate in order to reduce the losses. We have used Adaptive Moment Estimation (ADAM) optimizer [19] similar to Voyager [10].

VI. Drishyam in Action

Next, we describe how images created are used to make predictions (Figure 9). The image is first sent into the patch partition layer. We use the patch dimensions of 2×2 with three color bands red, green, and blue. This means an image of dimension 16×16 is broken down into 64 patches. The output from the patch partition layer is, therefore, a matrix consisting of 64 entries of dimension 12. Each entry contains the pixel information of each patch. Since the patches are of dimensions 2×2×3, the entries are 12 dimensional. The image is then sent into the linear embedding layer, which linearly transforms the patch features into dimension C. In our case, C is 256. The image is, therefore, now in the form of a matrix of shape 64×256, which is an input to the SWIN transformer block.

The SWIN transformer block uses shifted window attention. In the next layer, the window regions are shifted, and attention is calculated locally within the newly shifted windows. This ensures that patches that were isolated or could not interact with each other in the first layer can now do so in this layer. It is important to note that each pixel in the input image is an embedding component of the individual page accessed in that window. This shifted window attention, therefore, is useful in the context of prefetching as we find out which embedding components are more useful for the individual patches. The SWIN transformer block keeps the shape of the matrix intact and provides this as input to the patch merging layer. This layer reduces the feature map size by 2X and increases the dimension of each patch by 2X. The output from this layer is a matrix of shape 16×512. These patches are then sent to the second SWIN Transformer block, which functions similarly to the first one. The output from this block remains a matrix of size 16×512.

The next layer is the global average pooling layer. It has a straightforward operation of calculating the average of each feature map obtained from the previous layer. This averaging operation reduces the dimension of the data. It is a preparation step for the final classification layer. The average operation ensures that the model is more robust to spatial translations, i.e., if a particular feature appears anywhere in the feature map, it will be "learned" through the average operation. The output from this layer is the dimension of a single feature map that is 512×1. Finally, this is provided to the final dense layer with *softmax* activation function. The number of nodes in the dense layer must be the same as that of the number of classes (P). *Softmax* assigns a probability to each class, and these probabilities sum up to 1. In simple terms, the output of dimension P×1(P is the number of unique pages or offsets accessed by the application) from the dense layer gives us the probability that the image belongs to a certain class. Here, the class is the page that will be prefetched given the image. We choose the page with the highest probability. A similar procedure is followed for offset selection, except that P is fixed at 64.

Note that, we train each benchmark (application) in isolation. Therefore, each benchmark uses its own weights. This is because the dimensions of the embedding layer depend on the number of unique pages in the benchmark, which varies across benchmarks.

VII. Evaluation

Simulator and benchmarks. We evaluate our prefetcher using an extended version of ChampSim [20], which was released with the 1st ML Prefetching Competition, co-located with ISCA 2021. ChampSim has been used by recent data prefetching championships and state-of-the-art non-ML data prefetchers [1]–[3]. Table II shows the parameters for our simulated memory hierarchy, which is similar to Intel Sunny Cove microarchitecture [21]. The prefetcher is situated at the last-level cache (LLC) that prefetch data into the LLC from the DRAM. We use irregular benchmarks from SPEC CPU 2006, 2017, and GAP benchmark suites [22], [23]. These are memory-intensive benchmarks and state-of-the-art non-ML prefetchers fail to provide a significant performance improvement. However, an oracle prefetcher provides at-least 10% performance improvement. This selection of benchmarks is the same as prior works [10]. For each benchmark, we use their respective representative sim point(s) for simulation.

Metrics. We evaluate Drishyam using the following metrics: prefetch coverage, prefetch accuracy, performance, training, and inference time. Performance is calculated based on normalized execution time compared to no prefetching. We also compare Drishyam with the state-of-the-art ML prefetcher, Voyager. Please note that the ML prefetchers that appeared in the 1st ML data prefetching championship are not high performing.

Training. To train our prefetcher, we use the last-level cache (LLC) accesses (similar to Voyager) for applications after running them through ChampSim. We use Tensorflow 2.11.0 [24] for the training process. For every access, the prefetcher provides an address to be prefetched. We train our prefetcher for the initial 30M instructions and use the trained prefetcher for prefetching for the rest of the sim-point (100s of millions of instructions) [22], [23]. We detect an application phase change using LLC accesses per cycle [25] and in case of a phase change, we re-train our prefetcher for 30M instructions. We simulate Voyager [10] as per the implementation provided [26] and the parameters mentioned in the paper. We use a Tesla GPU for training.

Accuracy and coverage. Figures 10 and 11 show prefetch accuracy and coverage with Voyager and Drishyam. On average, compared to Voyager, Drishyam improves the accuracy from 50.97% to 89.54% and coverage from 28.39% to 66.62%. Drishyam provides maximum accuracy of 95.8% for `cc` and maximum coverage of 88.8% for `xalancbmk`.

Performance, training, inference time. Figure 12 shows improvement in execution time compared to a system with

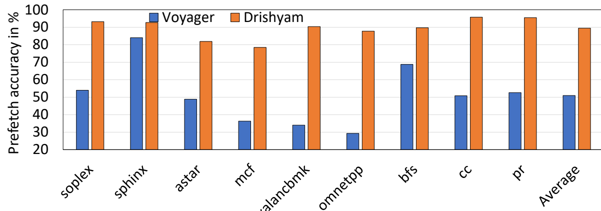

Fig. 10. Prefetch accuracy. Higher the better.

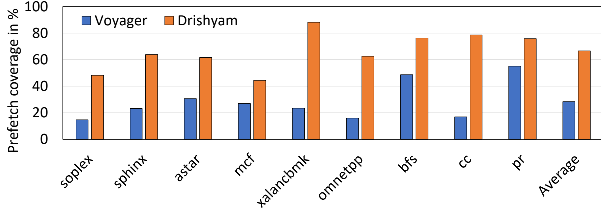

Fig. 11. Prefetch coverage. Higher the better.

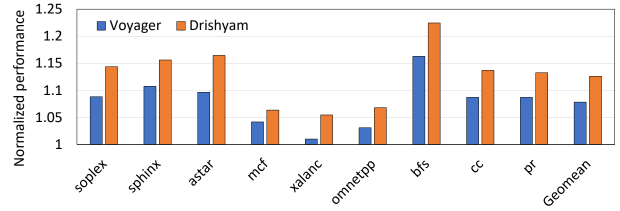

Fig. 12. Performance normalized to no prefetching. Higher the better.

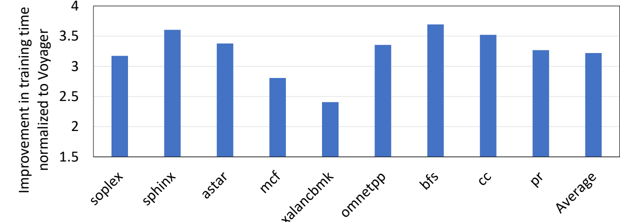

Fig. 13. Normalized training time improvement. Higher the better.

no prefetching. Drishyam provides an average performance improvement of 12.5%, whereas Voyager improves performance by 7.8% (4.7% less than Drishyam). Note that some of the benchmarks (like `xalancbmk` and `omnetpp`) show high prefetch coverage, but the performance improvement is about 5%. This is because the LLC misses per kilo instructions (MPKI) of these benchmarks are relatively low (just above one) as compared to `bfs` which has an MPKI closer to five.

Note that Voyager claims a performance improvement of 41.6% over a baseline with no prefetching. However, we report 7.8% over a baseline with no prefetching. There are two reasons for higher performance improvement: (i) the ChampSim simulator version that we use is more accurate with a detailed front-end and back-end. (ii) Voyager uses only one sim-point trace per benchmark. However, we use all the sim-points for a given benchmark. For example, `mcf` has nine sim-points [22] and Voyager uses only one of them. We corroborate the results of Voyager on the specific sim-points used in the Voyager paper.

Figure 13 shows training time improvement normalized to Voyager for the irregular benchmarks with their respective sim points. A value greater than one implies that training time is lesser than voyager. On average, Drishyam is 225.5% faster than Voyager. On average, Drishyam takes 1.9 minutes to train for one million instructions, whereas Voyager takes 6.26 minutes and as high as eight minutes for `mcf`. This is an expected trend as Voyager uses LSTM, which processes each word of a sentence separately, whereas transformers process the entire sentence in one go. In our case, sentences are the collection of pixels in the image, but for Voyager sentence means the sequences of memory addresses accessed. For inference, Drishyam is 20% faster than Voyager.

Storage overhead. Compared to Voyager, Drishyam's model size is 39.8% cheaper. Voyager incurs storage costs (in terms of storing the weights of different layers) of 76.8MB, whereas Drishyam incurs storage of 54.9MB. To understand the utility of this storage overhead, we compare the performance of Drishyam with a 64MB LLC per core. Figure 14 shows that on average, Drishyam outperforms a 64MB LLC (12.60% on average compared to 6.35%) as the reuse distance of LLC lines is extremely high and the working sets of these applications are in GBs (for example, mcf has a working set of 4GB).

Voyager and Drishyam. The core of Voyager's architecture is LSTM. In a nutshell, LSTM processes sentences, word by word. Whereas, Drishyam uses transformers and process a sentence as a whole, so Drishyam does not forget past information. Drishyam also uses shifted window attention to help in providing the relationship between words. In our case, the sentence is an image and words are the pixels in the image. We quantify this behavior for one of the benchmarks named `astar`. For the following sequence of pages, `38098601425`, `41059520803`, `38098601420`, `38098601420`, `43398299602`, `43398299603`, `43398299601`, `54301401950`, `38098601420`, `41059520781`, `38098601420`, `38098601426`, `38098601420`, `38098601427`, `54301401991`, `38098601420`. Voyager makes an incorrect page prediction and predicts `38098601420` whereas Drishyam makes a correct prediction and predicts `3927991042`. Overall, for this sequence, the confidence of prediction(probability that the sequence or image belongs to the predicted class) is low (0.10) for Voyager and high (0.74) for Drishyam.

Effect of revised offset embedding. Drishyam uses offset embedding that includes the page context and resolves the offset aliasing problem through attention. To see the impact of this revised embedding, Figure 15 shows overall accuracy, offset accuracy, and coverage with the original offset embedding (without page context). As we can see the impact of page context improves overall accuracy and coverage, significantly. Without page context, the accuracy and coverage reduce from 92.7% to 43.3% and 63.8% to 25.6%, respectively. This can be attributed to the offset accuracy, i.e. the percentage of prefetch requests where the offset is calculated correctly reduces from 94.4% to 46.1%. The offset accuracy reduction is because of offset aliasing.

Effect of the shifted window attention. Another important aspect of the prediction process is the image classifier. We

TABLE II
SIMULATION PARAMETERS OF THE BASELINE SYSTEM.

Core	Out-of-order, hashed perceptron branch predictor, 4 GHz with 6-issue width, 4-retire width, 352-entry ROB
TLBs	L1 iTLB/dTLB: 64 entries, 4-way, 1 cycle, STLB: 2048 entries, 16-way, 8 cycles
L1I	32 KB, 8-way, 4 cycles
L1D	48 KB, 12-way, 5 cycles
L2	512 KB 8-way associative, 10 cycles, non-inclusive
LLC	2 MB/core, 16-way, 20 cycles, non-inclusive, MSHRs: 8/16/32/64 at L1I/L1D/L2/LLC
DRAM	One channel, 6400 MTPS, FR-FCFS, 4 KB row-buffer per bank, open page, burst length 16

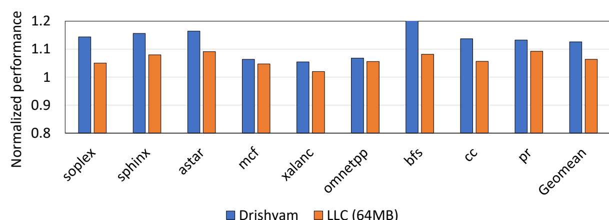

Fig. 14. Performance of Drishyam in comparison with a 64MB LLC normalized to an LLC with 2MB and no prefetching.

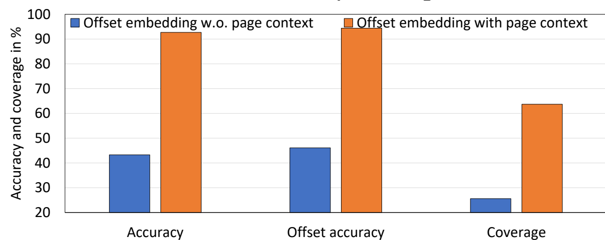

Fig. 15. Effect of page context in the offset embedding on performance.

have used the SWIN transformer as our image classifier. An advantage of using the SWIN transformer is its shifted window attention mechanism. To test the impact of this mechanism, we have tested a variant of the prefetcher with the Vision Transformer (VIT) [16] as the image classifier. VIT uses a global attention mechanism, i.e, each patch of the image is attended by all the other remaining patches.

The reason why the SWIN transformer works better than VIT is because its attention mechanism includes local attention within a window of patches and then another local attention after shifting the windows. This shifting procedure ensured that patches not within the same window in the first iteration are now in the same window. Since each patch represents embedding components for pages and offsets, because of the shifting mechanism, patches that were isolated could now be used for calculating the local attention between them. Figure 16 shows the utility of the SWIN transformer in terms of prefetch accuracy and coverage.

A possible hardware implementation. As mentioned in Section I, the goal of this paper is not to propose a practical implementation but rather to make a case of computer vision can also be used for prefetching. The hardware design for our prefetcher requires two major sub-components: image Creation and the SWIN transformer. The images are created using embeddings of individual pages accesses accessed. The embeddings are nothing but a hardware table with each table

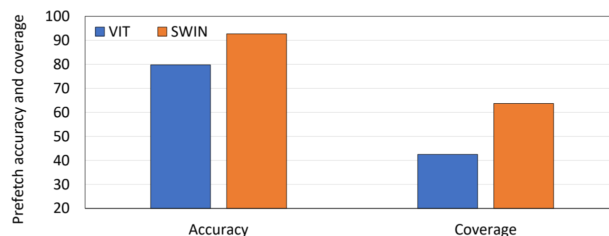

Fig. 16. Performance of the prefetcher with VIT and SWIN transformer.

containing the embedding entry for the pages. To calculate the embeddings, an extension of Bidirectional Encoder Representations of Transformers(BERT) [27] can be used. Compared to the SWIN transformer, which has an encoder and decoder, BERT only has an encoder.

We use these embeddings for creating images. Since the convention is that each pixel in an image has a value ranging from [1,255], we can create an alternate representation for these images. As each pixel in the image represents one component of the embedding vector for a particular page, we use these embedding vectors directly. In simple terms, if we consider an image length of 16 and an embedding vector of dimension 16, then the input to the SWIN transformer block will be of dimension 16×16. The next important sub-component of this prefetcher is the SWIN transformer model, which can be designed similarly to [28] with row-wise scheduling. Exploration of the practical aspects is beyond the scope of this paper, and it is a promising avenue for future research.

VIII. CONCLUSION

We make a case for the usage of computer vision for data prefetching, where we use images to visualize memory accesses. We use a SWIN transformer for predicting the future prefetch address by breaking the problem into page prediction and offset prediction. Our prefetcher, Drishyam, outperforms the state-of-the-art ML prefetcher for a set of irregular benchmarks in terms of accuracy, coverage, performance, training time, and cost. We believe the insights from this paper can motivate for practical implementation of Drishyam.

IX. ACKNOWLEDGEMENT

We would like to thank all the anonymous reviewers for their insightful comments and suggestions. Special thanks to Saradindu Kar, Kranti Kumar Parida, Debasish Das, and Alberto Ros for their feedback. This work is supported by the Qualcomm faculty award 2022 and Google India research award 2022.

References

[1] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. V. Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*, pp. 975–991, IEEE, 2022.

[2] 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)*, pp. 118–131, IEEE, 2020.

[3] 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)*, pp. 1–12, IEEE, 2016.

[4] 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)*, pp. 399–411, IEEE, 2019.

[5] 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)*, pp. 141–152, 2015.

[6] 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*, ISCA '15, (New York, NY, USA), p. 285–297, Association for Computing Machinery, 2015.

[7] L. Peled, U. Weiser, and Y. Etsion, "A neural network prefetcher for arbitrary memory access patterns," *ACM Trans. Archit. Code Optim.*, vol. 16, oct 2019.

[8] 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, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*, pp. 461–470, ACM, 2019.

[9] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018* (J. G. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 1924–1933, PMLR, 2018.

[10] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," ASPLOS '21, (New York, NY, USA), p. 861–873, Association for Computing Machinery, 2021.

[11] T. D. Doudali, "A picture is worth a thousand... features! using computer vision alongside machine learning in computer systems," *Wild and Crazy Ideas, ASPLOS 2022*, pp. 1–2, 2022.

[12] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, (Los Alamitos, CA, USA), pp. 9992–10002, IEEE Computer Society, oct 2021.

[13] S. Hochreiter and J. Schmidhuber, "Lstm can solve hard long time lag problems," in *Proceedings of the 9th International Conference on Neural Information Processing Systems*, (Cambridge, MA, USA), p. 473–479, MIT Press, 1996.

[14] "The 1st ml-based data prefetching competition," 2021.

[15] 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*, MICRO '21, (New York, NY, USA), p. 1121–1137, Association for Computing Machinery, 2021.

[16] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, OpenReview.net, 2021.

[17] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, "Semantic understanding of scenes through the ade20k dataset," *International Journal of Computer Vision*, vol. 127, pp. 302–321, 2018.

[18] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 740–755, Springer International Publishing, 2014.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

[20] "Champsim simulator." Available at https://github.com/ChampSim/ChampSim.

[21] "SunnyCove," May 2018.

[22] "Spec cpu traces." Available at https://dpc3.compas.cs.stonybrook.edu/champsim-traces/speccpu/.

[23] "Gap traces." Available at https://utexas.app.box.com/s/2k54kp8zvrqdfaa8cdhfquvcxwh7yn85/folder/132804598561.

[24] "Tensorflow." Available at https://github.com/tensorflow/tensorflow/releases.

[25] N. S. Kalani and B. Panda, "Instruction criticality based energy-efficient hardware data prefetching," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 146–149, 2021.

[26] "Voyager source code." Available at https://github.com/Quangmire/voyager.

[27] Z. Liu, G. Li, and J. Cheng, "Hardware acceleration of fully quantized bert for efficient natural language processing," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 513–516, 2021.

[28] H.-Y. Wang and T.-S. Chang, "Row-wise accelerator for vision transformer," 2022.