Building Efficient Neural Prefetcher

Yuchen Liu
Princeton University
Princeton, USA
yl16@princeton.edu

Georgios Tziantzioulis
Princeton University
Princeton, USA
georgios.tziantzioulis@pm.me

David Wentzlaff
Princeton University
Princeton, USA
wentzlaf@princeton.edu

ABSTRACT

Data prefetching is a promising approach to mitigate computation slow down due to the memory wall. While modern workloads grow more and more complicated, their memory access patterns become less organized and rule-based prefetchers can no longer deliver improved performance, which motivates the research of adopting neural networks for prefetching. However, current neural prefetchers require high computation costs and large storage space to obtain good performance, which makes them far from practical. To this end, we address the efficiency issue in neural prefetchers and propose an effective approach to build light-weight models. Specifically, our method is aware of both machine learning and micro-architecture, where we introduce a novel neural prefetcher design space with knobs from both aspects. We optimize these knobs using workload characteristic observations, rigorous mathematical optimization, and efficient design space traversal, which provides us with highly-efficient neural prefetchers. Our approach is evaluated on SPEC CPU 2006, where our models can provide up to 60% IPC gain compared to no prefetching, outperforming non-neural based prefetchers. In comparison with the state of the art neural prefetcher, our models enjoy an average of 15.4× multiply-accumulation reduction, 6.7× parameters saving, with even better IPC gains. Although it is still challenging to provide implementable neural prefetcher, this order of magnitude computational and storage reduction, provided by our method, marks an important milestone towards practical neural prefetchers.

CCS CONCEPTS

• **Computer systems organization → Architectures**; **Processors and memory architectures**; • **Computing methodologies → Neural networks**.

1 INTRODUCTION

In the post Moore's Law era, memory accesses become orders of magnitude slower than processor computation with a large portion of cycles spent waiting for data to arrive from memory. This issue is widely known as the memory wall [49], which represents a critical bottleneck for modern computer architecture development. To combat the memory wall, a hierarchical memory system with caches is generally adopted, and several predictive caching techniques including data prefetching are introduced for better memory management.

Data prefetching aims to resolve the memory wall by hiding the long memory latencies of cache misses. Historically, architects have developed rule-based prefetching techniques by observations in memory strides [23, 42], instruction pointers [9, 38], temporal locality [20, 31, 48], and spatial locality [6, 24]. These methods implement fixed prefetching rules guided by heuristics, which are subject to a clear performance drop when applied on larger modern workloads [12].

While machine state transition provides the basis for data prefetching algorithms, it also naturally formulates prefetching as a time-series prediction problem, where deep neural networks (DNNs), from the machine learning community, have made major headway in this research domain. In fact, DNNs have shown promising results in large-scale sequential problems, e.g., text understanding [29], speech recognition [13], weather forecasting [52], and even stock market modelling [50]. Another feature of data prefetching, that an application provides millions to billions of memory access to feed DNN learning, makes DNN a favorable technique to solve this big data problem. Moreover, the rapid development of multiple neural processing units (NPUs) [4] improve the practicality of using DNN-based predictors for data prefetching from the architectural perspective.

A few prior works have attempted to use DNNs for data prefetching [12, 34, 41, 45, 53]. For example, Zeng et al. propose an LSTM model [53] to perform regression prediction for data prefetching. In a follow-up work, Hashemi et al. [12] propose a delta-LSTM to prefetch memory deltas, and they reform the objective to be closed-set classification rather than open-set regression. Zhan et al. [41] further address the class explosion problem in classification-based data prefetching and propose a hierarchical neural prefetcher which obtains state-of-the-art results. However, most of these works miss a critical aspect in their DNNs, the network efficiency, which is crucial for practical implementation. For instance, **Delta-LSTM [12] and Voyager [41] require around 40 million multiply-accumulation operations (MACs) to make a single prefetch prediction**, far exceeding the computational budget available to run in micro-architectures.

To this end, we pioneer to build efficient neural prefetchers with a better tradeoff between computational complexity and system performance. As shown in Figure 1, our design reduces the inference cost over an order of magnitude from the state-of-the-art neural prefetcher [41], achieving an even better IPC improvement for the system.

We start our investigation by characterizing the inference path and profiling the computational cost of modern neural prefetchers to understand where there is opportunity to enhance efficiency. Specifically, we find that most MACs in a neural prefetcher occur in three computation phases: embedding, network, and prediction. Based on that, we develop a novel design space with crucial knobs from both perspectives of machine learning model design and micro-architectural configuration, where knobs in such space directly relate to the complexity-performance tradeoff. With a set of design knobs, the computation graph of a hierarchical neural prefetcher model is fully specified, and the state-of-the-art model [41] can also be expressed under our search space.

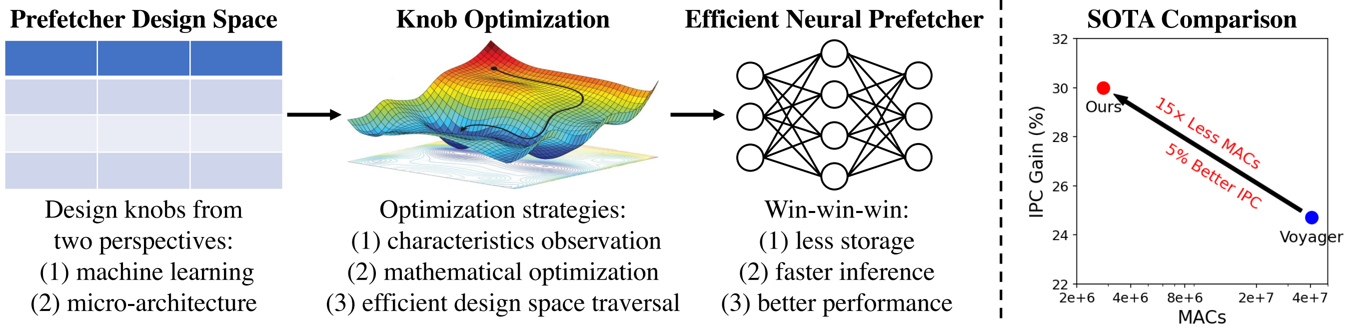


Figure 1: We outline our method to build highly-efficient neural prefetchers and show a sample of our results. Specifically, we propose a design space with knobs for both ML and micro-architecture and develop multiple strategies to optimize these knobs. The derived prefetcher is highly efficient with less storage, faster computation, and even better performance. Compared to the state-of-the-art neural prefetcher [41], we achieve order of magnitude computation reduction (horizontal axis in log scale) with a higher IPC gain on a memory-intensive benchmark, `mcf`.

While the vast space contains a total number of 7.2×10^{11} prefetcher designs, we present multiple strategies to help us find optimal knobs for efficient prefetchers. Firstly, we optimize the knobs by characteristic observation, where we analyze the characteristic of computation workloads as well as neural network topologies and conduct empirical experiments to understand their effectiveness. With such strategy, we study two knobs that are ignored in most prior works but in fact crucially important to its efficiency: (1) the data stream that the ML model should learn on; (2) the alternative recurrent neural network (RNN) structure which could be more efficient than widely-adopted LSTM [14]. Our second strategy is to lay out mathematical optimization objectives to rigorously derive the desired knobs. We adopt this strategy for determining the bit position to split data memory addresses for hierarchical prediction. While a page-offset configuration naturally determines a split position for memory addresses, we find that naively adopting this setting could yield redundant prefetchers with weaker performance. We thus propose a novel objective function to find the optimal address split by considering two quantified metrics: (1) the computational complexity of the prefetcher (2) and distribution randomness of the split address. Finally, we develop a low-cost design space traversal strategy to search for efficient knobs. Specifically, we establish a step-by-step method to optimize the knobs for vector dimensions, including sizes of embedding vectors, the number of hidden neurons, and the history length to feed into a recurrent network. To save the search cost, we sub-sample the knob design space and evaluate a knob's effectiveness with all other knobs frozen. Our result show that this intuitive strategy can already provide us with highly efficient networks.

Our contributions are three-fold:

(1) We present a study that enhances the efficiency of neural prefetchers with a novel design space. Specifically, this design space is derived via a characterization of the inference path and a profiling of the compuatition cost on modern prefetchers, from which we identify crucial efficiency-impactful knobs for both aspects of machine learning and micro-architecture.

(2) We propose effective strategies to optimize the design knobs to help search for efficient prefetchers under the vast space. Our strategies involve observing the characteristics of the neural networks and benchmark workloads, mathematical optimization for the memory address split, as well as a low-cost algorithm to quickly traverse the design space. With these effective strategies, we can discover sets of knobs that build up efficient prefetchers.

(3) On SPEC CPU 2006, our models show a maximum of 58.3% IPC gain compared to no prefetching. On a subset of memory-intensive benchmarks, `libquantum`, `omnetpp`, `mcf`, and `xalancbmk`, our models provide an average IPC gain of 33.7% and 26.3% compared to no prefetching and generic non-learning based prefetcher. Compared to the state of the art neural prefetcher, our models enjoy a 15.4× MAC reduction and a 6.7× parameter saving on all benchmarks, with an average of 3.2% IPC gain on memory-intensive benchmarks, leading the state of the art.

2 BACKGROUND

2.1 Neural Network for Micro-Architecture

Applying machine learning to solve micro-architecture problems has long been attempted. Previously, reinforcement learning (RL) techniques are used to optimize memory controller scheduling algorithms [16], as well as performance knobs [7].

Neural networks, a widely-studied model in the machine learning community, have also been applied to multiple micro-architecture tasks [10, 21, 30, 40, 46]. They have demonstrated strong prediction capabilities especifically for tasks of branch prediction [21, 46] and cache replacement [40]. Jimenez et al. [21] proposes a single-layer perceptron to learn dynamic branch prediction rules, while Shi et al. [40] use an LSTM to distill information into a support vector machine (SVM) classifier for cache replacement.

While branch prediction and cache replacement are binary prediction problems (taken vs. not taken and keep vs. replace), with light-weight models like one-layer perceptron producing desirable results, data prefetching represents a much harder multi-value prediction challenge, and the state-of-the-art prefetchers [12, 41] require gigantic DNNs to model more complicated patterns and larger prediction space. Although DNNs provide significant better prediction power than prior rule-based approachs [5, 20], it is still

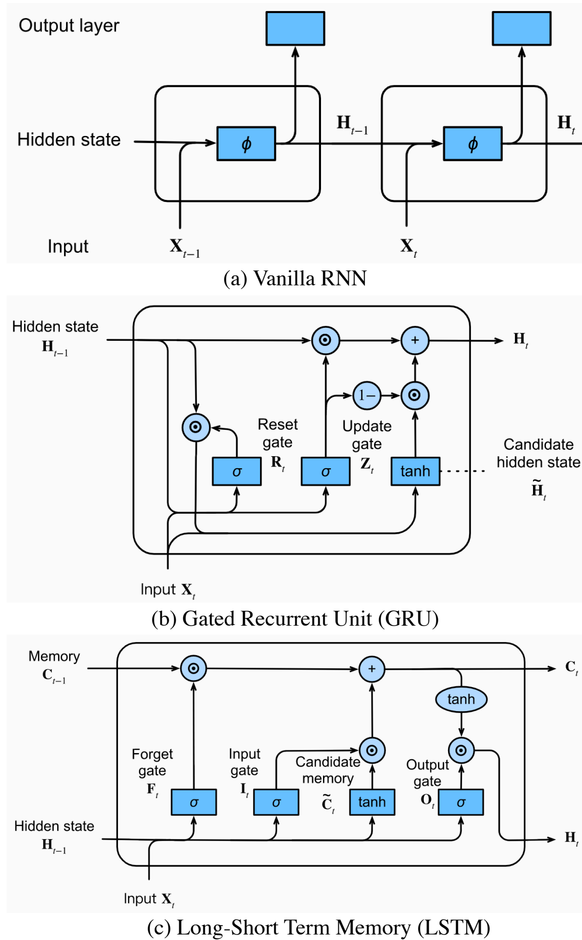


Figure 2: Different RNN structures as shown in a textbook [54].

hard to incorporate them into a real system. Thus, enhancing the efficiency of a neural prefetcher is of critical importance.

2.2 Recurrent Neural Network

Recurrent neural networks (RNNs) are a type of modern DNNs with recurrent connections, which excels in time-series modelling. Since most micro-architecture prediction tasks incur machine state transitions which spontaneously form sequences of time steps, RNNs naturally become the first model to try among different DNNs. For data prefetching, LSTM [14] has become the dominant RNN variant to build into a neural prefetcher. However, although LSTMs show strong modelling capability for the tasks, it may not be the optimal choice when network efficiency is taken into consideration.

In fact, there are multiple variants of RNNs shown in Figure 2 including (a) vanilla design [35] (b) gated recurrent unit (GRU) [8] and (c) long short-term memory (LSTM) [14]. These RNN variants adopt a common structure where the block at time step t will have an input of data embedding $\mathbf{X}_t$ and the hidden representation from the prior block $\mathbf{H}_{t-1}$ to compute its own representation $\mathbf{H}_t$ which will be recurrently served as the input of the (t+1)-th block. However, even if we fix $\mathbf{X}$ and $\mathbf{H}$ to have the same dimensionality, these variants will require different amount of computation. This is because they have different designs and numbers of internal gates within a block to generate $\mathbf{H}$, and computing the activations for each gate requires a matrix multiplication transforming the input vector into the gate vector. Precisely speaking, the vanilla RNN only has 1 internal gate while GRU has 3 gates and LSTM has 4 gates. That being said, switching LSTM to a GRU/vanilla design could save 25%/75% of computation cost without affecting the representation dimensionality. In our work, we find the choice of RNN structure to be a crucial knob to enhance prefetcher's efficiency.

2.3 Efficient Neural Networks

Efficiency has become a critical issue in the neural network community with increasing attempts in building efficient neural models by structural pruning [25, 27], matrix factorization [19], knowledge distillation [13, 26], and neural architecture search (NAS) [55, 56]. Among them, structural network pruning and NAS lead the design of efficient networks as they produce compact and dense models that are compatible with the Basic Linear Algebra Subprograms (BLAS) libraries [25]. In general, structural network pruning aims to remove redundant neurons from a gigantic baseline model, while NAS defines a network search space and leverage optimization techniques like reinforcement learning (RL) agents to find good specifications of models.

Our work bears a similar flavor to pruning and NAS, as it aims to reduce computation cost with a reference baseline and creates a design space to traverse by optimizing different knobs. However, our approach has significant novelties: (1) While these works are mostly proposed to build efficient DNNs for computer vision problems, we make the first attempt to incorporate these techniques to solve the crucial mirco-architecture prediction challenge of data prefetching, to the best of our knowledge; (2) Prior techniques only aim to optimize ML metric like prediction accuracy, yet the architectural performance is more accurately reflected by IPC, and hence our design space includes architectural knobs in addition to ML knobs; (3) Instead of using an expensive controller and evaluating on millions of candidate networks to search for the best one as done in prior NAS work, our strategy requires much less search cost to discover an efficient design.

3 PROBLEM FORMULATION

To improve the efficiency of a neural prefetcher in both dimensions of complexity and performance, we need to first characterize its inference path and profile how much computation is incurred for each phase in this path. This characterization and profiling allows us to analyze which part of the prefetcher is potentially redundant and reducible as well as where we possibly have a chance to improve performance.

3.1 Neural Prefetcher Computation Phases

Most modern neural prefetchers share a common inference path shown in Figure 3, where the inference mainly involves three phases: embedding, network, and prediction. (1) In embedding, the prefetcher takes a history stream of read memory accesses

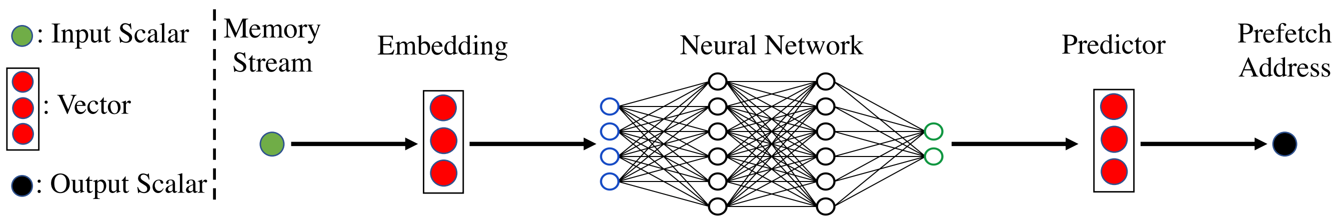


Figure 3: Overall flowchart of a neural prefetcher. The scalars in the memory stream are first embedded into high dimensional vectors, which are then fed to a neural network for representation learning. The learned representations will be used for predicting the prefetch address.

(full/miss memory stream) as input with scalar values of program counters (PCs) and memory addresses. These scalars are then used as keys to retrieve their high-dimensional vector embeddings from a pre-stored lookup table. (2) In network, the prior embeddings are then fed into a neural network, typically a recurrent neural network like LSTM, for a series of non-linear operations to output a vector representation. (3) In prediction, the output representation is fed into a predictor to predict the prefetch address. The predictor usually uses a single linear transformation to either approximate the address value by regression or treat each unique address as a class and perform class probability prediction.

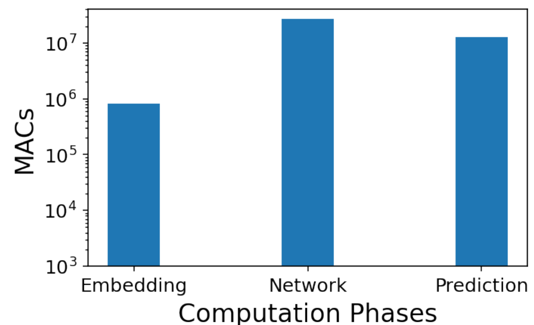


Figure 4: Computational cost of Voyager [41] on `mcf`. The vertical axis is logarithmic.

3.2 Computation Profiling

Following the phase characterization in Figure 3, we profile the computational cost of the state-of-the-art neural prefetcher, Voyager [41], running on `mcf` in Figure 4. We observe that there is large room for efficiency enhancement in each phase. (1) In embedding, Voyager embeds PCs and addresses into vectors with too high dimensions that some entries of the vectors could be redundant. Moreover, it uses a number of attention experts [47] for correlation embedding, where some of the experts are unnecessary and can be safely removed. (2) In the network, Voyager adopts LSTM as its network structure which has multiple gates to generate its hidden representation. While LSTM does have good performance for time-series modelling, there are other light-weight RNN structures like Vanilla RNN and GRU that bear similar performance. In addition, some neurons in the network may not be useful and can thus be pruned. Moreover, the computation cost of an RNN scales linearly with the length of input entries, and thus using a shorter history of PCs/memory accesses would help reduce the network's cost. (3) For prediction, the computational cost can actually be as high as the network phase. A noticeable issue of a classification-based neural prefetcher is the class explosion problem, where a workload may have millions of unique addresses and hence unique classes which requires a gigantic linear predictor. Voyager attempts to solve this issue by splitting an address into two portions, page and offset, where the number of unique pages and offsets is orders of magnitude less than the unique addresses. However, this default split still creates a large number of classes to predict on some benchmarks which is cumbersome and would deteriorate the performance.

4 OUR SOLUTION

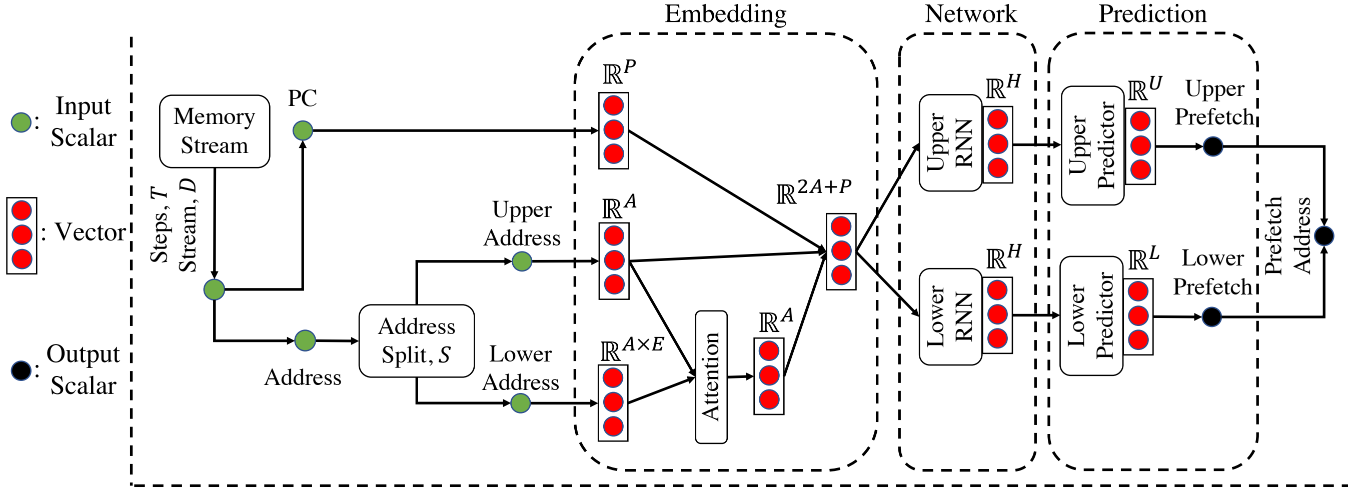
4.1 Neural Prefetcher Design Space

Inspired by the inference path characterization and computation cost profiling in Section 3, we introduce our novel design space for a neural prefetcher with parameterization knobs shown in Table 1. Specifically, our design space includes knobs from both perspectives of micro-architecture and machine learning design, with the types and ranges of values for each knob included. Being aware of the computational efficiency, we incorporate what computation phases (embedding, network, and prediction) these knobs would affect.

With a specification of knobs, we are able to translate the parameterization into the actual computational graph of a neural prefetcher shown in Figure 5. Specifically, the prefetcher takes in a history length of T PCs and addresses from a memory stream D, where the addresses are split into two portions with the upper parts containing (64-S)-bit addresses and the lower parts S-bit. The scalar values of PCs and two parts of the addresses are embedded into vectors with knobs of P, A, and E specifying their dimensions, where we adopt the same attention mechanism [47] as in [41] to obtain the final embedding of lower addresses. The embeddings of PCs and both parts of the addresses are concatenated and fed into two identical RNNs of type N with the same number of H neurons to learn their representations. The learned representations are later fed into two classifiers to predict the upper prefetch (64-S)-bit addresses and lower S-bit prefetch addresses, which can be finally combined into 64-bit prefetch addresses.

Perspective	Name	Knob	Type	Range	Computation Phase
Architecture	stream dataset	D	str	{Full, L1DM, L2CM, LLCM}	embedding & prediction
	address split	S	int	[12, 20]	embedding & prediction
ML Design	PC embedding	P	int	[1, 64]	embedding
	address embedding	A	int	[1, 256]	embedding
	attention experts	E	int	[1, 100]	embedding
	time steps	T	int	[1, 16]	embedding & network
	hidden neurons	H	int	[1, 256]	network
	RNN structure	N	str	{Vanilla, GRU, LSTM}	network

Table 1: Our design space for neural prefetcher.



1. All knobs are shown at the position that it will affect the computation graph.
2. The dimensionality of all vectors are notated.
3. U & L denotes the number of unique upper and lower addresses which are fully determined by D and S.

Figure 5: Computational graph of our neural prefetcher specified by the design knobs.

Our design space contains a total of 7.2×10^{11} unique neural prefetcher designs, each of which can be determined by a specification of *D-S-P-A-E-T-H-N*. For example, the state-of-the-art neural prefetcher Voyager [41] is a candidate network under the design space with the specification of L2CM-12-64-256-100-12-256-LSTM. Importantly, our design space includes several unique knobs that most prior works ignore but are found to be crucial for efficiency: (1) D, on what stream the neural prefetcher should train on and where the prefetcher should be placed to best integrate the ML model into the system; (2) S, where to split the addresses could be benchmark dependent and we need a systemic approach to determine a good splitting point; (3) N, while LSTMs are useful, they are computationally expensive and seeking other variants of RNN designs could alleviate the computation burden.

From Section 4.2 to Section 4.4, we discuss more details of our design knobs and lay out our strategy to optimize these knobs for a much more efficient neural prefetcher.

4.2 Training on the Right Memory Stream D

A neural prefetcher is a neural network, which naturally uses the metric of ML accuracy to evaluate its prediction power. A neural prefetcher is also a data prefetcher, whose ultimate goal is to enhance the system performance, typically measured by the IPC gain. *Does a prefetcher with high ML accuracy spontaneously delivers high IPC gain?* The answer is not necessarily yes. While the full memory stream, including all read memory accesses of misses and hits from all hierarchies, provides the most amount of data which is the best from the ML perspective for model learning and indeed gives the best ML accuracies, the neural prefetcher trained on it may not help much for the system performance. As shown in Figure 9, a neural prefetcher that makes 81.6% correct prediction on the full address stream turns out to have a lower IPC gain compared to another one that has 52.8% prediction accuracy for the L2 cache miss stream.

In fact, determining which memory streams to use as the dataset (D) for the neural prefetcher to train on is an open question and is a critical block for building an efficient neural prefetcher. In this work, we investigate four memory read streams for knob D: full

Benchmark	Full	L1D Miss	L2C Miss	LLC Miss
astar	29M	0.56M	0.37M	0.31M
bzip2	25M	1.04M	0.32M	0.09M
gcc	20M	0.76M	0.16M	0.05M
gobmk	22M	0.40M	0.15M	0.04M
h264ref	33M	0.27M	0.13M	0.06M
hmmer	44M	0.21M	0.02M	5.74K
libquantum	15M	2.86M	2.86M	2.86M
mcf	29M	12M	8.76M	7.52M
omnetpp	23M	2.71M	2.32M	1.83M
perlbench	22M	0.37M	0.21M	0.17M
sjeng	25M	0.07M	0.05M	0.05M
xalancbmk	22M	3.61M	3.60M	3.32M

Table 2: Number of memory accesses for different streams over the SPEC06 benchmark suite with the simulator design in Table 3. The number is shown taken from Pinpoints [32] of 130M instructions.

stream (Full), L1D miss stream (L1DM), L2C miss stream (L2CM), and LLC miss stream (LLCM). We also study two types of prefetcher placement, in-place prefetching and LLC prefetching. For in-place prefetching, we train and put the neural prefetcher on the same level of cache hierarchy. For example, a prefetcher trained on L1DM is put on the L1D cache (the prefetcher trained on Full is placed at LLC). For LLC prefetching, we follow the scheme as in ISB [20] and Voyager [41], where the prefetchers are learned on a lower-level cache (L1D, L2C) and placed into the LLC. The reason to do LLC prefetching is that the prefetches would not disrupt the future miss stream in the lower caches if we treat a prefetch as load. However, if we maintain a separate prefetch queue for each cache, in-place prefetching would not have the issue of stream disruption either.

We present the statistics of different D in Table 2 for the benchmark suite of SPEC2006 prior to training. In general, the lengths of the miss streams are orders of magnitude less than the their full counterpart. Moreover, we find two groups of benchmark with similar cache behavior: (1) the miss streams of each cache hierarchy are roughly the same or on the same order (e.g., omnetpp, mcf); (2) the lengths of miss streams reduce by orders of magnitude when it goes to an upper cache hierarchy (e.g., bzip2, gcc). Preferably, the choice of D should deliver good performance for both cache behaviors.

4.3 Optimizing Address Split S by Complexity and Randomness

As shown in the computational graph of Figure 4, our prefetcher is divided into an upper address RNN predictor and one for the lower addresses. For that, we propose a knob S to allow flexibility in determining where to split the memory address into two portions.

Formally, we denoted the memory address stream M (fully determined by D), which is separated into an upper stream M_U and a lower stream M_L, where each entry in M_U is a (64-S)-bit integer and M_L contains S-bit integers. As we are simulating a system with 64-byte cache lines, the last 6 bits for an entry in M_L will not affect the prefetching performance, and hence we can further alter M_L to a stream with (S-6)-bit integers. In our design, we use an identical RNN structure N with the same number of hidden neurons H for both M_U and M_L prediction. To derive efficient prefetchers, we optimize S based on the principle of computation complexity and distribution randomness by extracting statistics from M_U and M_L.

Complexity. The numbers of unique sub-addresses in M_U and M_L, denoted by U and L, are fully defined by S. The computational complexity of the prediction phase (referred to Figure 5) is thus $H \times (U + L)$ MACs. Moreover, we observe that $U \times L$ is close to a constant across different S, for all benchmarks, as U and L are sort of uncorrelated. Due to that, if U is much larger than L, we can simply increase S to reduce U and raise L such that the new pair of (U, L) will have a smaller sum for a smaller computation complexity cost. When L is much larger than U, we can decrease S following the same notion. In fact, when $U \times L$ are constrained to be a constant, the minimum of $U + L$ is derived when $U = L$ on the real number space, and thus we would like U and L as close to each other as possible to minimize their sum. In our design, U and L are determined by S, where we do not have the freedom to choose U and L to be any number. We therefore introduce a quantitative metric to evaluate how close U and L are given an S as follows:

$$C(S) = |\log_2(U) - \log_2(L)| \tag{1}$$

where C is a function of S, denoting the discrepancy of computation complexity between M_U and M_L, and minimizing C would give us an S that makes the predictor the most efficient. Note, we add a $\log_2$ in front of U and L, as U and L generally scale/decay by 2 when S is increase or decrease by 1, and putting them in $\log_2$ would facilitate the incorporation of other terms that do not scale exponentially.

Randomness. A prefetch address is only correct when we make correct predictions for both its upper (64-S) bits and its lower (S-6) bits. Because of that, the final prefetch accuracy is bounded by the more challenging stream to predict between M_U and M_L. Since we use an identical RNN structure (same N and H) for both representation learning of M_U and M_L with the same network capacity, we would like M_L and M_U to be equally hard to predict, which would minimize the maximum hardness between them.

The question then becomes how to measure the predictive hardness for M_U and M_L without conducting the expensive neural network training. Our solution is to approximate that by using the concept of distribution randomness. In fact, M_U and M_L can also be seen as two data distributions, where the occurring frequency for each sub-address can be treated as its probability in each distribution. The more random the distribution is, the harder to make a prediction on the value of it. Drawing from information theory, the randomness of any data distribution X can be quantified by its entropy, h[1]:

$$h(X) = \sum_x p(x) \log p(x) \tag{2}$$

where x is the possible value that X can take. With the entropy well defined, we can then introduce a term to quantitate the discrepancy between M_U and M_L in their distribution randomness, which is a good approximation of their predictive hardness:

$$R(S) = |h(M_U) - h(M_L)| \tag{3}$$

[1]Conventionally, entropy is denoted by H. Since we have already use H as one of our design knob, we use h to denote the entropy.

where R is a function of S and minimize R would result in a choice of address split that makes M_U and M_L equally hard for representation learning.

Optimization Objective. Formally, our joint optimization objective to choose S based on the computation complexity and distribution randomness can be formulated as:

$$\min_{S} \ \alpha C(S) + \beta R(S) \tag{4}$$

where α and β are two positive weights for each term. Although in theory S can be chosen as any integer from 6 to 64, in practice we find that the optimal S for Equation 4 always lies between 12 and 20 which could save a lot of computation costs for S optimization.

4.4 Neural Architecture Search

After investigating D and S which are crucial parameters from the micro-architecture perspectives, we proceed to the rest of the knobs which are more related to ML design. How to design an efficient neural network for a task is always an open problem. The network structure to adopt, the dimension for each of the vectors, and the number of hidden neurons in a network can all be optimized for more efficient design. Indeed, searching for an efficient neural architecture is highly non-trivial.

RNN Structure. LSTM is the most commonly adopted RNN variant for neural prefetcher [12, 34, 41, 45, 53], due to its strength in modelling time-series and the existence of multiple gates. However, while each gate requires H hidden neurons, the use of multiple gates could increase the computational cost. Although the design of gates in LSTM is theoretically sound, some of the gates could be redundant and we could be able to reduce them in practice.

In our study, we have an N knob (referred to Table 1) for the structure of RNN. In fact, with the same number of hidden neurons (H) in the learned representation, the vanilla RNN and GRU only require 1 and 3 gates, which are both less than 4 gates in LSTM. That being said, using a vanilla RNN and GRU can have 75% and 25% complexity savings compared to using an LSTM, while maintaining the dimensionality for the learned representation. As no prior works attempt to address the efficiency issue of the network for prefetching, we pioneer an empirical study to choose the type of RNN structure from the standpoint of finding a better performance-complexity tradeoff.

Dimensionality Search. The rest of the knobs, P, A, E, T, and H, decide the dimensionality of input/hidden high-dimensional embedding vectors. In general, the smaller the values are, the less computation cost the prefetchers will have. In our design space, the choice of these values could have $64\times256\times100\times16\times256 = 6.7\times10^9$ combinations, which is impossible to enumerate.

To traverse such a vast design space, we adopt a straightforward approach where we gradually search for each knob one by one and subsample the range of values to reduce the search cost. For example, when optimizing the PC embedding dimension P, we freeze all other knobs without tuning them, and sub-sample the design space from [1, 64] to {32, 64} and evaluate the network performance on all values in the sub-sampled space. From an efficiency perspective, we choose the smallest knob that maintains the performance above a predetermined threshold. The knob optimization loop of P is thus complete and we can proceed to the next unoptimized knob, e.g. A for the next optimization. In practice, we perform the dimensionality knob search in the order of $T \rightarrow P \rightarrow A \rightarrow E \rightarrow H$, and we only need less than 20 evaluations to traverse the vast 6.7×10^9 possibilities which represents a low-cost search algorithm.

Processor	352-entry ROB, 4-wide, 128-entry LQ, 72-entry SQ, 128-entry scheduler bimodal branch prediction
L1 I-Cache	32KB, 8-way, 4-cycle latency, 8 MSHR
L1 D-Cache	48KB, 12-way, 5-cycle latency, 16 MSHR
L2 Cache	512KB, 8-way, 10-cycle latency, 32 MSHR
LLC per core	2MB, 16-way, 20-cycle latency, 64 MSHR
Memory	tRP=tRCD=tCAS=20 2 channels, 8 ranks, 8 banks 32K rows, 8GB/s bandwidths

Table 3: ChampSim simulator design.

5 EVALUATION

We first discuss our evaluation methodology in Section 5.1. We then compare our efficient neural prefetcher with various state-of-the-art prefetchers [6, 18, 24, 41], both neural and rule-based, in Section 5.2. Specifically, when compared to the state-of-the-art neural prefetcher [41], we achieve noticeable MAC reduction and parameter saving with even better performance. Lastly, we describe how we reach the efficient design with step-by-step ablation studies in Section 5.3-5.6.

5.1 Experimental Methodology

Simulator. We use the open-sourced ChampSim simulator [1] to validate the system performance of our efficient neural prefetchers in a micro-architecture. ChampSim models a single-core 4-wide out-of-order processor with an 8-stage pipeline and a three-level cache hierarchy. We set the parameters specifically to model an Ice Lake micro-architecture [2] as shown in Table 3 for the workload.

Benchmarks. We conduct our experiment over the full suite of SPEC CPU 2006 [3] (SPEC06). For reproducibility, we use the set of open-sourced SPEC06 PinBalls[2]. For each SPEC06 benchmark, we simulate all PinPoints [32] and aggregate their performance statistics based on the methodology introduced in [11]. While each PinPoint contains 100M warmup instructions and 30M region of interest (ROI) instructions, this provide us a natural split for the neural prefetcher learning where we use the 100M warmup instructions for training and the 30M ROI instructions for testing. Similar to [41], our method can be adopted for online training where the prefetchers are trained and tested on a different portion of the benchmark's execution.

Training Neural Network. In addition to the design knobs which specified the inference path of the prefetcher, we adopt the training recipe in Table 4 to train all the models. In fact, the recipe is the same as the one in Voyager [41] **which reflects that our novelty and effectiveness stems from the design space instead of the training hyperparameters**.

[2]http://snipersim.org/w/Pinballs

Hyperparameters	Values
Training epoch	50
Batch size	512
Learning rate	0.001
Number of RNN layers	1
Dropout keep ratio	0.8

Table 4: Recipe for training our neural prefetchers.

Compared Methods. We mainly compare against the state-of-the-art neural prefetcher, Voyager [41]. In particular, we adopt their open-sourced implementation[3] to reproduce their results on our trace. We use the same training recipe in Table 4 for Voyager to ensure the fairness in the comparison.

Evaluation Metrics. Since our prefetcher can be placed on different levels of cache hierarchy than Voyager, we do not use the common hierarchy-specific metrics like accuracy and coverage. Moreover, we find that the ML prediction accuracy, unified accuracy/coverage, does not help much in identifying the efficacy of a prefetcher as shown in Section 5.3. Therefore, we solely report the effectiveness of the prefetcher on the overall system performance by IPC. To measure computational complexity, we measure the total MACs for one inference of a prefetcher to evaluate its computational budget. We also include the number of parameters that a prefetcher needs in order to evaluate its storage requirement.

5.2 Comparison to the State of the Art

We compare our models with various arts, including Signature Path Prefetcher (SPP) [24], Access Map Pattern Matching (AMPM) prefetcher [18], Bingo [6], and Voyager [41], where our method advances the state-of-the-art IPC. When compared to the state-of-the-art neural prefetcher, Voyager [41], we outperform it in both complexity and performance.

Complexity. As shown in Figure 6 and 7, our efficient neural prefetchers provide an average reduction of 15.4× in multiply-accumulation operations (MACs) and an average parameter saving of 6.7× compared to Voyager [41], which represents much more light-weight models.

Performance. In Figure 8, we show the IPC gain percentages of different prefetchers compared to no prefetching. While all prefetchers provide similar performance on benchmarks that are not memory-bounded, our efficient neural prefetcher demonstrates a significant advantage on a subset of memory-intensive benchmarks: {`libquantum`, `mcf`, `omnetpp`, `xalancbmk`}. Specifically, in comparison to generic non-learning based prefetchers of SPP [24] and AMPM [18], our model provides an average of 16.6% and 26.3% IPC gain on these memory-intensive benchmarks. Compared to the state-of-the art neural prefetcher, Voyager [41], our model shows an average of 3.2% IPC gain on the memory-intensive benchmarks, advancing the state of the art.

5.3 Choosing Data Stream D

We now show how we derive our efficient design in 5.2 starting with knob D. Our first investigation is shown in Figure 9, where we train

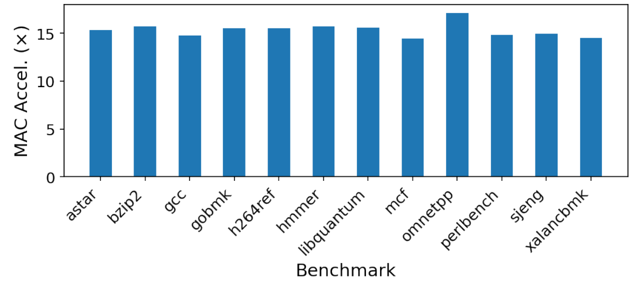


Figure 6: MACs acceleration compared to Voyager [41]over the entire suite of SPEC06. On average, our model enjoys a 15.4× MACs acceleration.

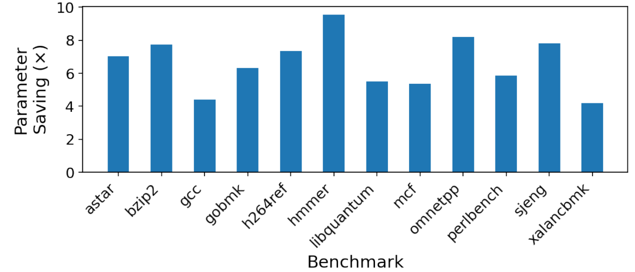


Figure 7: Parameters saving compared to Voyager [41]over the entire suite of SPEC06. On average, our model enjoys a 6.7× parameter saving.

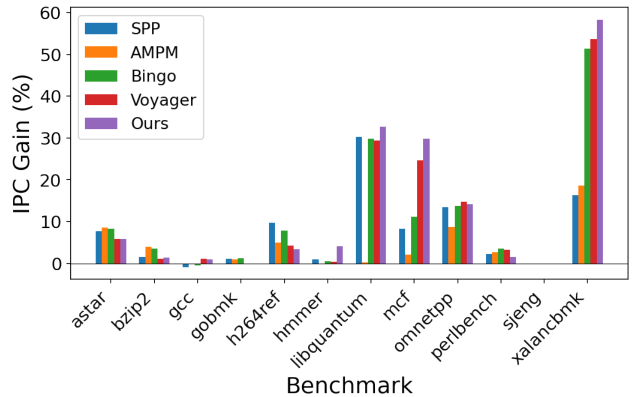


Figure 8: IPC gain compared to no prefetcher on the entire suite of SPEC06 with both methods. On memory intensive benchmarks with more cache misses (`libquantum`, `mcf`, `omnetpp`, and `xalancbmk`), our method shows an average of 33.7% IPC gain over no prefetcher, and 3.2% gain compared to Voyager [41].

with Full and L2CM streams of `omnetpp` and adopt corresponding in-place prefetching to evaluate the IPC gain for both settings. Although the prefetcher produces a much higher ML accuracy on Full, the one trained on L2CM achieves much higher IPC gain. This indicates that a good ML metric, like accuracy, does not ensure a

[3]https://github.com/aleczhanshi/neural_hierarchical_sequence

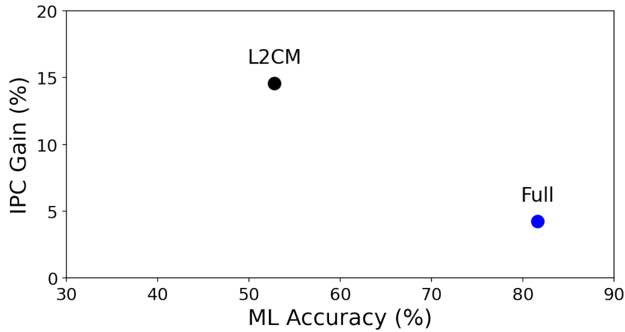


Figure 9: Would training on the stream with the most amount of samples (Full) provide us with the most powerful prefetcher? The answer is yes from the ML perspective but probably no from the perspective of architecture CPU performance. We find that achieving a good ML accuracy on the full stream would give less IPC gain than predicting on the L2 miss stream with less accuracy, which reflects the importance of knob D for neural prefetcher design. Experiments conducted on `omnetpp`.

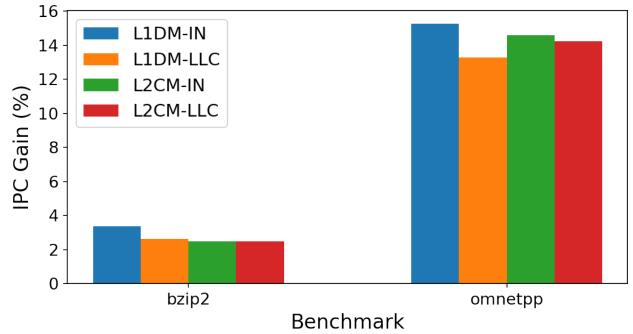


Figure 10: In-place prefetching on different D. Models trained on L1DM achieve the best IPC gain.

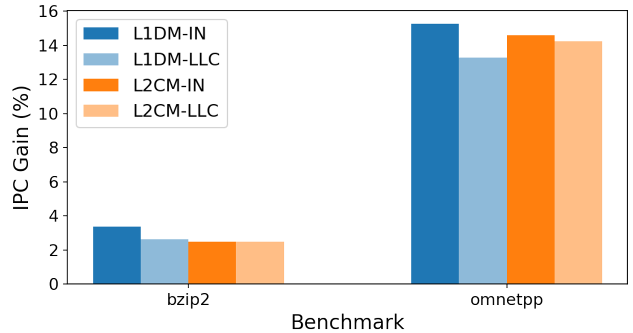


Figure 11: In-place vs. LLC prefetching for prefetcher trained on lower level caches. Our choice of in-place prefetching on L1D (L1DM-IN) delivers around 1.5% IPC gain compared to state-of-the-art neural prefetcher design [41], L2CM-LLC, without any the ML components.

good system performance, which reinforce that building an efficient neural prefetcher should not only consider ML factors but also the micro-architectural design with crucial parameters like D.

Next, we conduct a comprehensive study on how we should choose D for a neural prefetcher. According to Table 2, we select one representative benchmark for each group of cache behavior: `omnetpp` for group (1) and `bzip` and group (2). We first train the neural prefetcher for in-place prefetching as shown in Figure 10. Although Full contains the most amount of data which provide better results for neural network learning accuracy, we find that the prefetchers trained on it yields the least IPC gain for both types of benchmarks. Moreover, we find that training on L1DM and then placing the prefetchers at the L1D cache consistently produces the best performance for both benchmarks. We further compare in-place prefetching vs. LLC prefetching for models trained on lower-level caches and show the results in Figure 11. While Voyager [41] is trained on L2CM and placed at the LLC (L2CM-LLC), we find that doing L1DM in-place prefetching (L1DM-IN) can deliver around 1.5% IPC gain without changing the ML components. We thus fix L1DM in-place prefetching as our design knob of D in the following study.

5.4 Optimizing Address Split S

After D, the next knob we investigate is S where we use the objective in Equation 4 to optimize it. Specifically, we use a memory intensive benchmark, `mcf`, to perform the study and set α=β=1. As shown in Figure 12, the optimization objective shows a nice convex landspace where it reaches its global optimal when S=18 for `mcf`. In fact, such convexity applies to all benchmarks in SPEC06 suite and we find the optimal S vary between [14, 18] instead of the default page-offset setting of S=12. Moreover, as we make our actual implementation for the neural prefetcher, we find that the optimal split does provide us with a light-weight model (Figure 13) with top performance (Figure 14), which has higher efficiency than the default split. This result shows that our formulation from the perspective of computation complexity and distribution randomness is mathematically rigorous and empirically effective.

5.5 Choice of RNN Structure N

After fixing D and S, we look into the structural type of RNN, N, with `mcf`. As shown in Figure 15, we find that LSTM indeed provides a good IPC gain due to its strong modelling effectiveness, GRU can also perform on par with it while having a 25% MAC reduction for free. This shows that adopting GRU for N could be more efficient while prior works do not consider it. Moreover, using Vanilla RNN does reduce MACs from LSTM by an order of 3, yet it also exhibits a noticeable performance drop. Hence, we choose GRU over the vanilla design in our study.

5.6 Design Space Traversal

Lastly, we present a design space traversal from Voyager [41] to our prefetcher on `mcf` with step-by-step knob optimization shown in Figure 16. Being consistent with the results in Section 5.3, 5.4, and 5.5, optimizing the knobs of D (step 1), S (step 2), and N (step 3), gives us win-win models with both less complexity and better

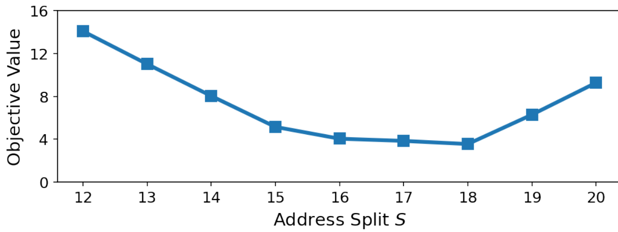


Figure 12: The objective value of Equation. 4 over different S on benchmark `mcf`. The optimal S for the objective value is 18.

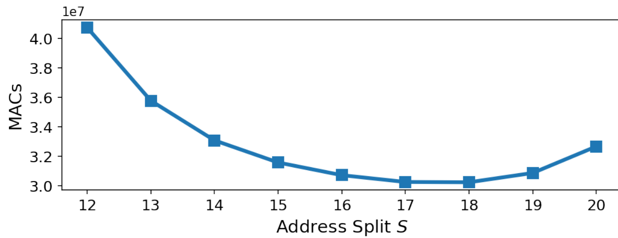


Figure 13: Models' MACs with different S on benchmark `mcf`. The most light-weight model corresponds to the optimal split determined by Equation. 4.

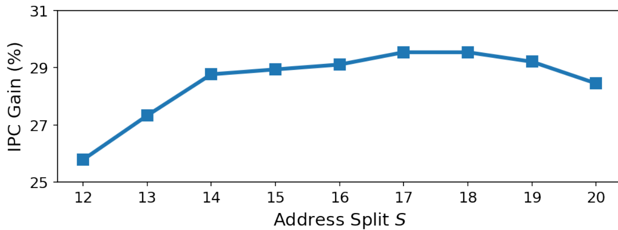


Figure 14: Models' IPC gains with different S on benchmark `mcf`. The optimal split determined by Equation. 4 delivers a top IPC gain.

system performance. More specifically, we find that address embedding, A, is also a crucial knob to optimize (step 6), where less dimensionality on A makes the prefetcher easier to capture the pattern. The conventional pruning of hidden neurons is demonstrated in step 8 for optimizing H. While we perform the train from scratch instead of pruning from large, we think that to identify redundant nodes by pruning could even better enhance the model efficiency. With this traversal, we derive a model with 15× fewer MACs while 5% better IPC gain.

6 RELATED WORK

6.1 Heuristic-Based Prefetchers

Memory accesses usually show some visible patterns which are exploited to build prefetchers based on the stride difference [17, 23, 28, 36, 42], temporal locality [5, 15, 20, 22, 31, 48], and spatial locality [6, 24, 39, 43, 44, 51].

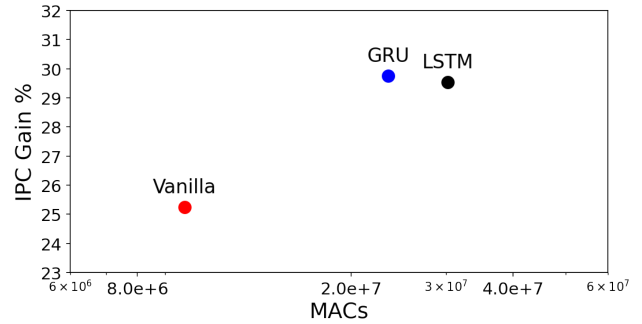


Figure 15: The efficiency of adopting different types of RNNs structure. As adopting GRU provides a 25% MACs reduction from LSTM while having even better IPC gain, we adopt it for our prefetcher design. Experiments are conducted with `mcf`, and the horizontal axis is logarithmic.

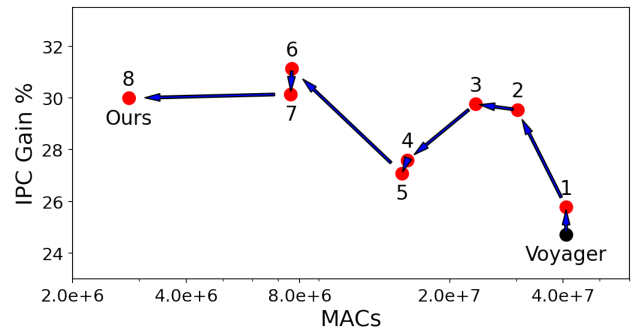


Figure 16: Step-by-step knob optimization design space traversal from Voyager [41] to our 15× efficient design. The knob adjustments for each step: (1) D: L2CM → L1DM; (2) S: 12 → 18; (3) N: LSTM → GRU; (4) T: 16 → 10; (5) P: 64 → 32; (6) A: 256 → 32; (7) E: 100 → 30; (8) H: 256 → 128. Experiments are conducted with `mcf`, and the horizontal axis is logarithmic.

In stride prefetching, a stream buffer [23] is adopted to observe if a constant stride occurs over a sequence of consecutive requested memory addresses. The earliest prefetcher of this type is the classic next-line prefetcher [42] which is used for concept introduction in most textbooks. Recently, stride prefetchers are led by Sandbox [36] and best-offset prefetcher [28], which determine the optimal stride based on the coverage after testing a few candidates.

Researchers later develop prefetch schemes based on the temporal locality of memory addresses with an original work formulating the prefetcher problem as a Markov decision process [22]. This direction is further enhanced by the GHB [31] prefetcher, which uses a sequence of non-repeating access history as a lookup table for decision. STMS [48] further sample the memory stream to reduce the overhead of the meta-data, and a more recent work of Domino prefetcher [5] uses the two past addresses in the global stream instead of only looking at the last one to make the prefetch.

In addition to temporal locality of absolute addresses, spatial patterns such as locally-repeated address deltas or address offsets are

incorporated as a factor to determine the prefetch address. A typical work is SMS [44] prefetcher, which applies spatial patterns seen in the old pages to new pages. The VLDP [39] prefetcher records the history of address deltas for varied-length memory accesses. This work is followed up by SSP [24], which defines a signature function based on address deltas to output the next address delta. The Bingo [6] prefetcher further extends the address contexts to increase the precision of regional pattern matching.

Compared to these methods, our work allows the prefetcher to automatically discover and flexibly adjust the temporal and spatial patterns by showing them historic examples to a learnable neural network, rather than by adopting fixed rules.

6.2 Learning-Based Prefetchers

Another line of prefetcher designs adopts machine learning approaches to learn the rules. Early work in [37] leverage logistic regression and decision tree models for prefetch prediction, and a reinforcement learning table-based approach is introduced in [33].

More recently, DNN prefetchers [12, 34, 41, 45, 53] show more promising results as the large network capacity allows the prefetcher to capture more complicated and diverse memory behaviors. Specifically, Zeng et al. [53] and Peled et al. [34] adopt LSTM networks to perform regression tasks for prefetch prediction. While training regression LSTMs allows prefetchers to approximate the values of prefetch addresses, it is less desired since a prefetch line that even only differs by a small amount from the ground truth would not be useful. In contrast, we adopt classification LSTMs for our prefetcher for more precise prediction. In this line, an expensive delta-based classification LSTM prefetcher [12] is previously proposed and a follow-up work [45] claims to reduce the complexity of final prediction layers from $O(n)$ to $O(\log n)$ at the best case. However, these prefetchers still bear a high inference cost and they do not incorporate their prefetchers into a simulator that can report the overall system performance gain in IPC. On the contrary, our prefetcher is orders of magnitude smaller and we provide a study with practical simulation on system performance to demonstrate our effectiveness.

Our work is most related to Voyager [41] which uses a hierarchical neural prefetcher that reduces the final predictor complexity to $O(n/64)$ based on a page-offset address split, and achieves state-of-the-art performance-compelxity tradeoff. Different from that, we present multiple design knobs that are ignored in Voyager. With effective knob optimization strategies, our neural prefetchers are 15.4× less costly with even better performance compared to Voyager, leading the state of the art.

7 CONCLUSION

In this work, we propose a novel mechanism to build efficient neural prefetchers. Specifically, we introduce a design space with efficiency-impactful knobs after characterizing the inference path and profiling the computation cost of modern neural prefetchers. These knobs include both perspectives of machine learning design and architectural configuration. Then, we introduce strategies to optimize these knobs via characteristic observation, mathematical optimization, and low-cost design space traversal algorithm. These strategies allow us to quickly derive efficient models. When evaluating on SPEC CPU 2006, our method provides up to 60% IPC gain compared to the no prefetcher setting, outperforming non-neural based prefetchers. In comparison with the state-of-the-art neural prefetcher, we achieves 15.4× computation acceleration, 6.7× parameter saving with better IPC. With the growing power of on chip neural-accelerator, our method could lead to more practical neural prefetcher with the state-of-the-art efficiency.

REFERENCES

[1] [n. d.]. *ChampSim*. https://github.com/ChampSim/ChampSim.

[2] [n. d.]. *Ice Lake Microprocessor*. https://en.wikipedia.org/wiki/Ice_Lake_(microprocessor).

[3] [n. d.]. *SPEC CPU 2006*. https://www.spec.org/cpu2006/.

[4] [n. d.]. *WikiChip*. https://en.wikichip.org/wiki/neural_processor.

[5] 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.

[6] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 399–411.

[7] Ronald D Blanton, Xin Li, Ken Mai, Diana Marculescu, Radu Marculescu, Jeyanandh Paramesh, Jeff Schneider, and Donald E Thomas. 2015. Statistical learning in chip (SLIC). In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 664–669.

[8] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[9] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A stateless, content-directed data prefetching mechanism. *ACM SIGPLAN Notices* 37, 10 (2002), 279–290.

[10] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 449–460.

[11] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[12] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *International Conference on Machine Learning*. PMLR, 1919–1928.

[13] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, and Brian Kingsbury. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine* 29, 6 (2012), 82–97.

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

[15] 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.

[16] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 39–50.

[17] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd international conference on Supercomputing*. 499–500.

[18] 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.

[19] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866* (2014).

[20] 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.

[21] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 197–206.

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

[23] 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.

[24] 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.

[25] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).

[26] Yuchen Liu, Zhixin Shu, Yijun Li, Zhe Lin, Federico Perazzi, and Sun-Yuan Kung. 2021. Content-aware gan compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12156–12166.

[27] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*. 2736–2744.

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

[29] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. Makuhari, 1045–1048.

[30] 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.

[31] Kyle J Nesbit and James E Smith. 2004. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 96–96.

[32] Harish Patil and Mack Stallcup. 2014. PinPoints: Simulation Region Selection with PinPlay and Sniper. *ISCA tutorial* (2014).

[33] 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.

[34] Leeor Peled, Uri Weiser, and Yoav Etsion. 2019. A neural network prefetcher for arbitrary memory access patterns. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–27.

[35] Fernando J Pineda. 1987. Generalization of back-propagation to recurrent neural networks. *Physical review letters* 59, 19 (1987), 2229.

[36] 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.

[37] Saami Rahman, Martin Burtscher, Ziliang Zong, and Apan 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*. IEEE, 383–389.

[38] Amir Roth and Gurindar S Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture*. 111–121.

[39] 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.

[40] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 413–425.

[41] 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.

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

[43] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 69–80.

[44] 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.

[45] 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.

[46] Elvira Teran, Zhe Wang, and Daniel A Jiménez. 2016. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[48] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 79–90.

[49] 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.

[50] Pengfei Yu and Xuesong Yan. 2020. Stock price prediction based on deep neural networks. *Neural Computing and Applications* 32, 6 (2020), 1609–1628.

[51] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. 178–190.

[52] Mohamed Akram Zaytar and Chaker El Amrani. 2016. Sequence to sequence weather forecasting with long short-term memory recurrent neural networks. *International Journal of Computer Applications* 143, 11 (2016), 7–11.

[53] 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.

[54] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. 2021. Dive into deep learning. *arXiv preprint arXiv:2106.11342* (2021).

[55] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[56] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.