Copyright
by
Zhan Shi
2020

The Dissertation Committee for Zhan Shi
certifies that this is the approved version of the following dissertation:

Machine Learning for Prediction Problems in Computer Architecture

Committee:

Calvin Lin, Supervisor

Don Fussell

Qiang Liu

Milad Hashemi

Machine Learning for Prediction Problems in Computer Architecture

by

Zhan Shi, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2020

Dedicated to my mom and dad.

Acknowledgments

My PhD experience has been a valuable journey for me, and for all that I have learned in the past years, I have many people to thank.

First and foremost, I am grateful to my advisor, Calvin Lin. It took around 2 weeks for me to realize that Dr. Lin is more of an educationist than a pure researcher or professor. Throughout my academic journey with Dr. Lin, I enjoy the respect and freedom that all my fellow students have been envying about. Dr. Lin patiently taught me the way to make impacts in the computer architecture community, and ensured that I learn from failures, and helped me see the bigger questions behind our research efforts. Also, I especially appreciate his thorough feedback on my presentation and writing, which has helped me organize my thoughts more logically and systematically.

I would also like to thank my committee members for their valuable feedback over the course of my PhD. Dr. Fussell has provided kind words of encouragement and important feedback on numerous occasions. Dr. Liu has given me very constructive comments on my RPE and proposal, and pointed out the directions that improved my work.

I have to give special thanks to my external committee member, Milad Hashemi. Milad has reshaped my Ph.D. through our 3-year collaboration, during which he has not only acted as my unofficial co-advisor, but also been a older

brother who always gave me suggestions, shared his own experience, and shown me the way to do research and be a good person. I'm so lucky to have worked with Milad in my Ph.D..

I also need to thank Akanksha for being a role model. From her I know what a successfully student of Dr. Lin could be like, which is always motivating. I am thankful to my group members, Hao, Kai, Curtis, Jia, Molly, Chirag, Aparna, and Jack, my collaborators Kevin, Danny, Partha, and my colleagues at Google and Facebook, Ashish, Guangsha, Yuening, Ehsan and many others, for their support, feedback, and great company. Last but not the least, I am fortunate to have a great support system in my family and friends. My parents have supported me unconditionally throughout my graduate education. My girlfriend, Judy has provided tremendous encouragement and support. I will never forget the time when I realized that I needed to get my proposal done in 2 weeks, from writing, to a couple of practices, to the final talk. Judy believed more than I did that I could make it in time, and took the family responsibility during COVID without any complaints. Thank you for being my luck.

Machine Learning for Prediction Problems in Computer Architecture

Publication No. _____

Zhan Shi, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Calvin Lin

The solutions to many problems in computer architecture involve predictions, which are often based on heuristics. Given the success of machine learning in solving prediction problems, it is natural to wonder if machine learning can better solve architectural prediction problems. Unfortunately, despite vastly outperforming traditional heuristics in various fields, machine learning has seen limited impact on prediction problems in computer architecture. The main challenge is that each architectural prediction problem exhibits unique constraints that prevent off-the-shelf machine learning algorithms from being more effective than heuristics. For example, hardware prediction problems, such as branch prediction and cache replacement, impose severe latency and area constraints that make multi-layer neural networks largely infeasible.

In this thesis, we propose machine learning solutions to three important problems in computer architecture, namely cache replacement, data prefetching,

and the automatic design of neural network accelerators. In our solutions, we focus on not only the design of learning algorithms, but also the use of learning algorithms under the unique constraints of each problem. In particular, to deal with the extremely tight area and latency constraints of replacement policies and data prefetchers, we propose to first design powerful yet impractical neural network models, from which we derive important insights that can be used to design practical predictors. To deal with the highly constrained search space in the automated design of neural network accelerators, we propose a new constrained Bayesian optimization framework to effectively explore the search space where over 90% of designs are infeasible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning has exploded in popularity for its ability to achieve state-of-the-art performance in a multitude of applications [75, 40, 85, 72]. These learning algorithms are capable of outperforming heuristic-driven approaches by extracting useful features in a data-driven fashion. Ideally, machine learning would be a powerful tool for computer architecture research, because heuristic-based predictions are commonly used in predictive mechanisms in speculative execution. For example, branch predictors [58, 103] predict a binary output—Taken or Not Taken—and cache replacement can be framed as a binary prediction—a line has either high or low priority for eviction [131, 52, 121]. Unfortunately, except for a few successes [58, 47, 59], machine learning has still not been widely adopted in hardware predictors [93].

The main challenges of applying machine learning are the unique constraints of each hardware prediction problem. For example, hardware cache replacement policies need to adapt to dynamic changes in the workload, so predictors for cache replacement are trained dynamically as the program executes. By contrast, deep learning models make multiple iterations over the training data and can take weeks or months to train. Moreover, the typical size of a hardware predictor for cache re-

placement is tens of bytes and a prediction must be produced in 20-30 CPU cycles (5-7 ns for a 4 GHz processor) [107, 1]. By contrast, deep learning models typically take megabytes to gigabytes of storage and microseconds to milliseconds to make a prediction.

Recent work shows the challenges of directly applying off-the-shelf machine learning to other architectural problems. For example, Tarsa et al. [120] and Zangeneh et al. [81] apply convolutional neural networks (CNNs) [72, 34] to branch prediction, which is another prediction mechanism that faces the tight area and latency constraints similar to cache replacement [57]. As shown in both papers [120, 81], although CNNs are capable of achieving superior accuracy, the models are too expensive to be deployed on chip. In particular, although a CNN with a unlimited budget shows the headroom of 2.9% IPC improvements over the state-of-the-art [103], the number shrinks to 0.6% when it comes to a practical CNN that fits on a chip area [81].

Similar issues apply to other hardware predictors, such as data prefetchers. In data prefetching, the predictor is not only constrained by tight latency, but also needs to learn correlations among tens of millions of unique address values. As a result, the large number of data addresses limits the accuracy of neural networks, which are designed for traditional tasks with orders of magnitude fewer unique values, such as natural language.

Beyond hardware predictors, machine learning could potentially be a powerful tool in the area of design automation of domain-specific accelerators. Design automation of neural accelerator relies on a cost model to predict the performance

and energy efficiency of possible designs, and the accuracy of the cost model determines the efficiency and effectiveness of the exploration process. However, due to constraints such as hardware budget, over 90% of hardware and software designs are invalid. As a result, the sparse and irregular search space leads to an inaccurate cost model that can severely hurt the efficiency of design space exploration.

The goal of this thesis is to find practical ways to use machine learning for hardware architecture problems under unique constraints and challenges. In particular, we first understand the root causes that prevent the effective use of machine learning techniques, and then we propose novel uses of machine learning that are specifically designed for each unique problem.

In the first part of this thesis, we focus on the extensively-studied problem of cache replacement. Cache replacement policies have evolved from ever more sophisticated heuristic-based solutions [102, 55, 118, 96, 3, 119, 22, 23] to learning-based solutions [130, 132, 52, 53, 121, 59]. We continue this trend by designing a powerful LSTM [41] learning model that can in an offline setting provide better accuracy than the state-of-the-art hardware predictors [52]. However, the offline LSTM exceeds the storage and latency requirements by 3 orders of magnitude. As the offline deep learning models is still not ready for direct use as hardware predictors, we need to understand why the offline LSTM provides accuracy improvements. To achieve this, we perform analysis to interpret this LSTM model with the scaled attention mechanism [80, 123], deriving a key insight that allows us to design a simple online model that matches the offline model's accuracy with orders of magnitude lower cost [107].

In the second part of this thesis, we focus on data prefetching, which is another extensively-studied memory optimization technique. Unlike previous neural models for prefetching that were limited to learning delta correlations [33, 117], we propose Voyager that can also learn address correlations, which are more powerful [50, 134, 133, 126, 5]. The major challenge of learning address correlations is that the number of data addresses are orders of magnitude larger than the number of unique categories in traditional machine learning tasks, such as natural language [80, 123]. The explosion of addresses not only leads to a significant increase in memory usage that could fail the training of neural networks, but also results in an ineffective representation of memory address space as each address only appears a few times, which is insufficient for training neural networks [33]. To solve this problem, we propose a hierarchical structure that separates addresses into pages and offsets [108]. The core of the hierarchical structure is a novel attention-based embedding mechanism for learning important relations among pages and offsets, and having pages provides contextual information for offsets. Though highly accurate, Voyager is still too expensive in computation to be practical for direct use in hardware. Therefore, we found two important insights that can lead to a practical prefetcher. First, the coverage of Voyager can be approximated with longest matching on two sequences of memory accesses, namely global sequence and PC localized sequence. Second, the order information is no longer important after we extract the longest-matched patterns, which helps us reduce the storage overhead by 33.6%.

In the third part of this thesis, we shift our focus from predictions problems

in hardware to the design of specialized deep learning accelerators. The compute and energy requirements of deep learning are growing [38], giving rise to numerous specialized hardware and software systems for deep learning. However, the design of such a system is typically driven by manual heuristics [17, 16, 21], or more recently heuristic-based search [135]. Therefore, we propose to cast the problem as hardware/software co-design, with the goal of automatically identifying desirable points in the joint design space [109]. The main challenge is that the highly constrained design space is semi-continuous / semi-discrete, meaning that a parameter can only take on a few discrete values, but their absolute values of the parameter affects the objective. The key to our solution is a new constrained Bayesian optimization framework that avoids invalid solutions through input and output constraints.

Thesis Statement. Machine learning can be a powerful tool in the solution of prediction problems in computer architecture, but to be effective, we need to devise new methods of tailoring machine learning techniques to these specific problems.

This thesis will make the following contributions:

1. We use cache replacement as a case study to explore the use of deep learning in hardware predictors that need to be dynamically trained and extremely efficient. Our solution is a three-step approach. In particular, we first build an offline deep learning model that achieves superior accuracy. In the second step, we derive the insights from the deep learning model that explains the advantage of deep learning models. In the last step, we use the insights to design the practical and effective Glider cache replacement policy that improves

the miss rate reduction over LRU from 7.1% to 8.9% in a single-core setting, and on a four-core system, improves IPC over LRU from 13.6% to 14.7%.

2. We apply our three-step approach to data prefetching, which is more challenging in all three steps. We first build a hierarchical offline deep learning model that can handle a vast memory address space. In particular, we build our domain knowledge into the neural model that separates addresses into pages and offsets, and we introduce a mechanism for learning important relations among pages and offsets. The proposed hierarchical neural model, which we refer to as Voyager, achieves an average IPC improvement of 41.6% over a system with no prefetcher, compared with 28.2% of prior art. At present, slow training and prediction still preclude Voyager from being practical in hardware. Therefore, in the second step, we derive insights of prefetching patterns, based on which comparable prefetching coverage can be approximated using tables without neural networks. We leave for future work to finish the third step for a practical design, which requires a smart management of tables.

3. For the automatic design of deep learning accelerators, we introduce a constrained Bayesian optimization framework that effectively handles the constrained feature space in which over 90% of design points are invalid. Our constrained Bayesian optimization framework is significantly faster and more robust than the constrained random search algorithm. As a result, our framework improves the energy-delay product (EDP) over the state-of-the-art ac-

celerator by 18.3% on ResNet, 40.2% on DQN, 21.8% on MLP, and 16.0% on Transformer.

This thesis is organized as follows. Section 2 places our work in the context of prior work. A detailed discussion and evaluation of the Glider replacement policy is presented in Section 3. Section 4 discusses the Voyager data prefetcher, and we introduce the constrained Bayesian optimization framework in Section 5. Finally, Section 6 concludes the dissertation.

Chapter 2

Related Work

We now place our work in the context of the considerable prior work of cache replacement, data prefetching and accelerator design respectively.

2.1 Cache Replacement

Replacement policies have evolved from heavily heuristic-based solutions to learning-based solutions. However, despite of its success in various fields, deep learning has not been used for hardware cache replacement.

Heuristic-Based Solutions. Most prior cache replacement policies uses heuristics to exploit commonly observed access patterns. The majority of prior work build on LRU, MRU and combinations of the two [111, 129, 76, 66, 27, 95, 102, 55, 118, 96, 42, 3, 119, 22, 23, 78, 68]. The rest of heuristic are based on frequency counters [99, 30, 69], re-reference interval prediction [55]. and the reuse distance [42, 3, 119, 22, 23]. A common downside of all heuristic-based policies is that they rely on human intuition to customize for a limited class of known cache access patterns.

Learning-Based Solutions. Recent advancement [67, 131, 52, 53] takes a learning-based approach that leverages the past caching behavior to predict future caching priorities. However, there has been almost no prior work that applies machine learning to the cache replacement problem. One notable exception [121] uses an online perceptron [100] to improve the accuracy of cache replacement predictors, but this solution sees only marginal improvements because it uses naive feature representations that result in a limited program context. More recently, Teran et al.'s perceptron was outperformed by MPPPB [60], which uses offline genetic algorithms to choose relevant features from a comprehensive list of hand-crafted features that go beyond control-flow information. Our work differs from MPPPB by identifying insights that lead to a more effective feature representation.

2.2 Data Prefetching

Prior work in data prefetching can also be described as either heuristic-based or learning-based. The vast majority of prefetchers are heuristic-based, meaning that they predict future memory accesses based on pre-determined patterns. Learning-based approaches, however, learn rules from the past access patterns and apply to the future execution.

Heurstic-Based Solutions. Many regular prefetchers predict sequential [112, 64, 44] or strided [89, 4, 25, 48, 101] streams by detecting constant strides in sequences of memory accesses. For example, stream buffers [64, 89, 4] confirm a constant stride if k consecutive memory accesses are the same stride apart. *Offset-*

based prefetchers [94, 82] improve upon these ideas by testing a few pre-determined strides to select an offset that provides the best coverage. Instead of predicting constant offsets, another class of prefetchers uses *delta correlation* to predict recurring delta patterns [88, 106]. Some irregular accesses can be captured by predicting recurring spatial patterns across different regions in memory [61, 73, 13, 14, 115]. For example, the SMS prefetcher [115] learns recurring spatial footprints within page-sized regions and applies old spatial patterns to new unseen regions, and the Bingo prefetcher [6] uses longer address contexts to predict footprints.

Temporal prefetchers learn irregular memory accesses by memorizing pairs of correlated addresses [19, 125]. Early temporal prefetchers correlated consecutive memory accesses in the global access stream [63, 87, 20, 114, 43, 127]. More recent temporal prefetchers look for correlations of consecutive addresses in a PC-localized stream [51, 134, 133]. This scheme improves coverage and accuracy due to the superior predictability of the PC-localized stream. Instead of using localization, Domino improve the predictability of temporal prefetchers by using two past addresses as contextual information [5].

A common limitation of all heuristic-based prefetchers is their use of simple known rules to identify the sequence of accesses, which limits the scope of patterns that can be detected. Recent work has also focused on improving the accuracy of aggressive stride prefetchers [10, 9].

Learning-Based Solutions. We now introduce previous learning-based prefetchers. Peled et al., use reinforcement learning to explore the correlation between pro-

gram contexts and memory addresses [92]. Their solution uses tables to learn a combinatorial number of context-address pairs, which leads to large storage overheads and slow training, and it precludes the learning of context-address pairs that do not repeat often. More recent solutions have instead used self-supervised learning to predict deltas, since it is easy to reduce the number of deltas to be smaller than than the number of addresses. For example, Hashemi et al. formulate delta prefetching as a classification problem and leverage LSTM neural networks [41] to find rich delta patterns [33, 110]. Recent work improves the efficiency of delta-based LSTM by compressing the input and output [117], but such work an undesirable tradeoff between coverage and efficiency.

2.3 Neural Network Accelerator

Different from the use of traditional CPUs, the design of hardware accelerators is often accompanied by a customized software optimizer that determines the use of software optimizations, such as look blocking and reordering. We now describe prior work in the hardware and software optimizations for DNNs.

Hardware Accelerators for DNNs. Prior work has designed specialized hardware to execute neural networks. From the design point of view, nearly all hardware accelerators are manually designed. Google's TPU [65] uses systolic arrays [74], and NVIDIA's GPUs have tensor cores [2]. Specialized for convolutional neural networks (CNNs), Eyeriss [17] introduces a specific dataflow that exploits a reuse pattern exhibited by 2D convolutions. To improve scalability, Eyeriss v2 [18] uses

a more sophisticated interconnect than its predecessor, and it also supports sparse CNNs. Prior work [91, 137] has dealt with sparsity by suppressing zero-valued activations and storing and operating on compressed data. Many other domain specific architectures have been proposed to take advantage of local communication patterns [24], 3D-stacked bandwidth memory [70, 28], or multi-chip modules [105].

Recent work [135] recognizes that the design space of specialized hardware is vast and proposes heuristics that can be leveraged to automatically synthesize hardware using a domain-specific language, Halide.

Software Optimization for DNNs. There has been considerable work on software optimizations for neural networks [128, 86, 11], which include optimizations, such as loop blocking (tiling), loop reordering, and loop unrolling, that affect the utilization of compute and storage resources. This optimization process has been recognized as a search problem, and compilers such as TVM [15] have used learned cost models to optimize execution efficiency. Similarly, Timeloop uses a grid or random search to optimize software mappings on a user-specified hardware architecture [90].

Chapter 3

Glider Cache Replacement Policy

3.1 Background and Constraints

Since our solution uses recurrent neural networks (RNNs) and attention mechanisms, we first provide background on hardware caches and these machine learning topics.

3.1.1 The Hawkeye Cache

This work builds on the Hawkeye cache replacement policy [52], which casts cache replacement as a supervised learning problem in which a predictor is trained from the optimal caching solution for past cache accesses.

Figure 3.1 shows the overall structure of Hawkeye. Its main components are OPTgen, which simulates the optimal solution's behavior to produce training labels, and the Hawkeye Predictor which learns the optimal solution. The Hawkeye predictor is a binary classifier, whose goal is to predict whether a line loaded by a memory access is likely to be cached or not by the optimal algorithm. *Cache-friendly* data is inserted in the cache with high priority, and *cache-averse* data is inserted with low priority.

Hawkeye uses the program counter (PC) as a feature and maintains a table

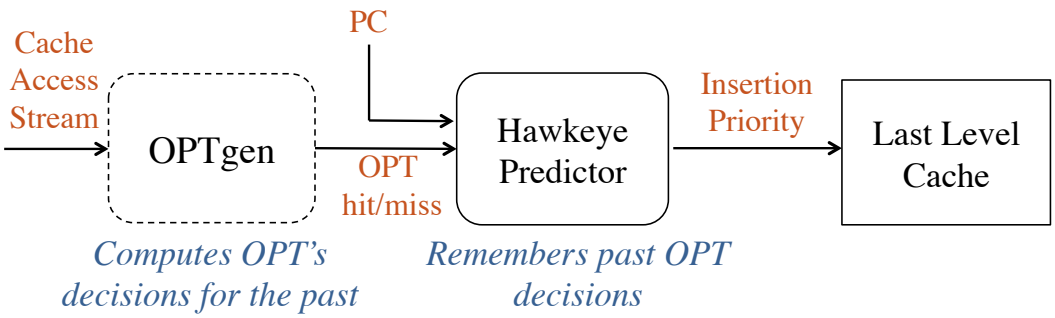


Figure 3.1: Overview of the Hawkeye Cache.

of counters to learn whether memory accesses by a given PC tend to be cache-friendly or cache-averse. While the Hawkeye Cache has been quite successful—it won the 2017 Cache Replacement Championship [1]—its simple predictor achieves only 72.4% accuracy on a set of challenging benchmarks.

3.1.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are extremely popular because they achieve state-of-the-art performance for many *sequential prediction problems,* including those found in NLP and speech recognition. Sequential prediction problems can make predictions either for the entire sequence (sequence classification) or for each element within the sequence (sequence tagging or sequence labeling). More technically, RNNs use their internal hidden states h to process a sequence, such that the hidden state of any given timestep depends on the current input and the previous hidden state.

LSTM is a widely used variant of RNNs that is designed to learn long,

complex patterns within a sequence. Here, a complex pattern is one that can exhibit non-linear correlation among elements in a sequence. For example, noun-verb agreement in the English language can be complicated by prepositional phrases, so the phrase, "the idea of using many layers" is singular, even though the prepositional phrase ("of using many layers") is plural.

We use LSTM because it has been successfully applied to problems that are similar to our formulation of caching, which we describe in Section 3.2. For example, part-of-speech tagging and name-entity recognition in NLP are both sequence tagging tasks that aim to assign a label to each element of a sentence.

3.1.3 Attention Mechanisms

LSTM has recently been coupled with *attention mechanisms,* which enable a sequence model to focus its attention on certain parts of the inputs. For example, when performing machine translation from a source language, say French, to a target language, say English, an attention mechanism could be used to learn the correlation between words in the original French sentence and its corresponding English translation [80]. As another example, in visual question answering systems, attention mechanisms have been used to focus on important parts of an image that are relevant to a particular question [79].

Mathematically, a typical attention mechanism quantifies the correlation between the hidden states h_t of the current step and all previous steps in the sequence using a scoring function which is then normalized with the Softmax function:

$$a_t(s) = \frac{exp(score(h_t, h_s))}{\sum_{s'} exp(score(h_t, h_{s'}))} \tag{3.1}$$

where $a_t(s)$ is the attention weight that represents the impact of the past hidden state s on the current hidden state t. Different scoring functions can be chosen [80], and the attention weights are further applied to the past hidden states to obtain the context vector:

$$c_t = \sum_s a_{ts} h_s \tag{3.2}$$

The context vector represents the cumulative impact of all past hidden states on the current step, which along with the current hidden states h_t form the output of the current step.

3.1.4 Hardware Constraints on the Predictor

There are three reasons why state-of-the-art machine learning models, including LSTM, are too resource-intensive to be deployed in hardware for problems such as cache replacement. First, the storage budget of hardware cache predictors is typically limited to 32 kilobytes [52, 121, 131]. Second, cache predictions must be generated in 20-30 CPU cycles (5-7 ns for a 4 GHz processor). Finally, the hardware predictor is required to converge after one training iteration (online training), as multiple training iterations increase latency, energy consumption, and storage.

In other resource-constrained applications of machine learning, online training is avoided by running inference on models that are pre-trained offline. However,

pre-training is difficult for cache predictions because (1) most features are categorical and each program have a unique vocabulary set. (2) unlike static code analysis, the dynamic behavior of one program can vary significantly across different inputs.

3.2 Our Solution

Our solution improves the accuracy of Hawkeye. Since Hawkeye learns from the optimal caching solution, improvements in the Hawkeye predictor's accuracy lead to replacement decisions that are closer to Belady's optimal solution, resulting in higher cache hit rates. Figure 3.2 shows the results of a limit study: We see that improving predictor accuracy leads to consistent improvements in program performance.

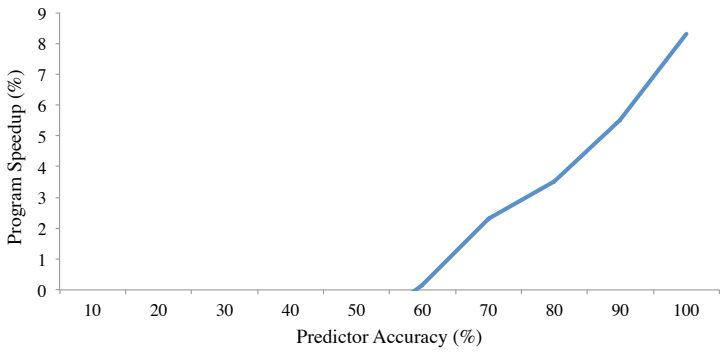


Figure 3.2: Improvements in predictor accuracy lead to increased speedup.

To improve predictor accuracy, we note that modern replacement policies [52, 67, 131], including Hawkeye, use limited program context—namely, the PC—to learn repetitive caching behavior. For example, if lines loaded by a given PC tend to be cache-friendly, then these policies will predict that future accesses by the

same PC will also be cache-friendly. Our work aims to improve prediction accuracy by exploiting richer dynamic program context, specifically, the sequence of past memory accesses that led to the current memory access. Thus, we formulate cache replacement as a *sequence labeling problem* where the goal is to label each access in a sequence with a binary label. More specifically, the input is a sequence of loads identified by their PC, and the goal is to learn whether a PC tends to access lines that are cache-friendly or cache-averse.

There are two reasons why we choose to identify loads by their PC instead of their memory address. First, there are fewer PCs, so they repeat more frequently than memory addresses, which speeds up training. Second, and more importantly, the size and learning time of LSTM both grow in proportion to the number of unique inputs, and since the number of unique addresses is 100-1000$\times$ larger than the typical inputs for LSTM [80], the use of memory addresses is infeasible for LSTM.

We now summarize our three-step approach:

1. **Unconstrained Offline Caching Model.** First, we design an unconstrained caching model that is trained offline (see Section 3.2.1). Our offline model uses an LSTM with an attention mechanism that identifies important PCs in the input sequence. We show that this model significantly outperforms the state-of-the-art Hawkeye predictor.

2. **Offline Analysis.** Second, we analyze the attention layer and discover an important insight: Caching decisions depend primarily on the presence of a few memory accesses, not on the full ordered sequence (see Section 3.2.2).

Thus, we can encode our input feature (the history of PCs) more compactly so that the important memory accesses can be easily identified in hardware by a simple hardware-friendly linear model.

3. **Practical Online Model.** Third, we use the insights from our analysis to build a practical SVM model that is trained online to identify the few important PCs; this SVM model comes close to the accuracy of the much larger and slower LSTM (see Section 3.2.3). The online version of this SVM model is essentially a perceptron.

3.2.1 LSTM with Scaled Attention

We now introduce our LSTM model that is designed for cache replacement. At a high level, our LSTM takes as input a sequence of load instructions and assigns as output a binary prediction to each element in the sequence, where the prediction indicates whether the corresponding load should be cached or not (see Figure 3.3).

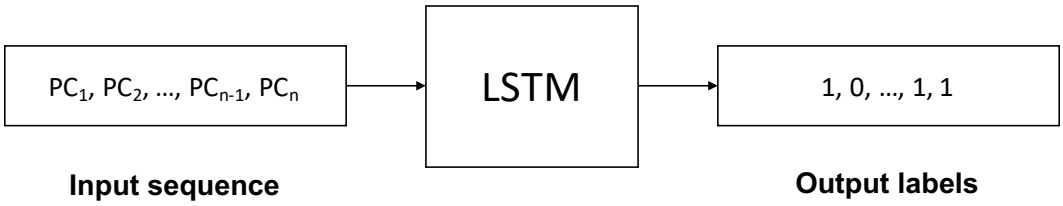


Figure 3.3: Our LSTM-based sequence labeling model takes as input a sequence of PCs and produces as output cache-friendly (1) or cache-averse (0) labels.

Figure 3.4 shows the network architecture of our LSTM model. We see that it consists of three layers: (1) an *embedding layer*, (2) a 1-layer LSTM, and (3) an attention layer. Since PCs are categorical in nature, our model uses a one-hot

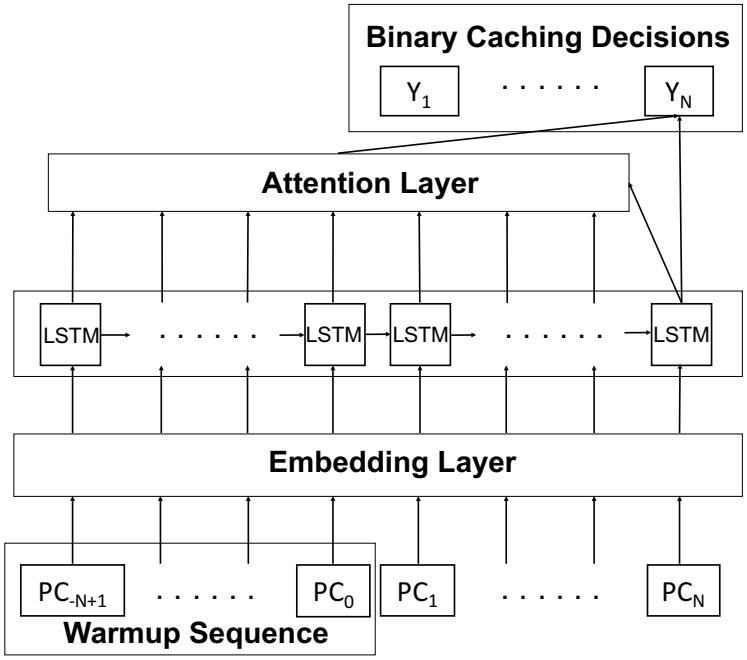


Figure 3.4: The attention-based LSTM network architecture. LSTM handles a sequence of input recurrently.

representation for PCs,[1] where the size of the PC vocabulary is the total number of PCs in the program. However, the one-hot representation is not ideal for neural networks because it treats each PC equally. So to create learnable representations for categorical features like the PC, we use an embedding layer before the LSTM layer. The LSTM layer learns caching behavior, and on top of the LSTM we add a scaled attention layer to learn correlations among PCs; we describe this layer in Section 3.2.2. Figure 3.4 shows the attention layer at time step N.

Since the memory access trace is too long (see Table 3.3) for LSTM-based models, we first preprocess the trace by slicing it into fixed-length sequences of

[1] A one-hot representation is a bit-vector with exactly one bit set.

Table 3.1: Statistics for benchmarks used in offline analysis.

Benchmark	# Accesses	# PCs	# Addrs	Ave. # Accesses per PC	Ave. # Accesses per Addr
mcf	19.9M	650	0.87M	30K	22.9
omnetpp	4.8M	1498	0.44M	3.2K	10.9
soplex	9.4M	2348	0.39M	3.9K	24.1
sphinx	3.0M	1698	0.11M	1.7K	27.3
astar	1.2M	54	0.31M	22K	3.8
lbm	5.0M	55	0.71M	90K	7.0

length $2N$. To avoid losing context for the beginning of each slice of the trace, we overlap consecutive sequences by half of the sequence length N. The first half of each sequence is thus a warmup sequence that provides context for the predictions of the second half of the sequence. In particular, for a memory access sequence PC_{-N+1}, ..., PC_N, the first half of the sequence PC_{-N+1} to PC_0 provides context of at least length N for PC_1 to PC_N. During training and testing, only the output decisions Y_1 to Y_N for time step 1, ..., N are collected from this sequence. Table 3.3.6 shows our specific hyper-parameters.

3.2.2 Insights from Our LSTM Model

Our LSTM model is effective but impractical, so we conduct several experiments to understand why our LSTM works well. First, we note that the LSTM's accuracy improves as we increase the PC history length from 10 to 30, and the accuracy benefits saturate at a history length of 30 (see Figure 3.15). This leads us to our first observation:

Observation 1. Our model benefits from a long history of past PCs, in particular, the past 30 PCs.

To gain even deeper insights into our attention-based LSTM model, we use the scaled attention mechanism to find the source of its accuracy. Our analysis of the attention layer reveals that while the history of PCs is a valuable feature, a completely ordered representation of the PC history is unnecessary. We now explain how we designed the attention layer so that it would reveal insights.

Attention Layer Design. Our attention layer uses the state-of-the-art attention mechanism with scaling [123] (see Equation 3.3). Our scaled attention layer uses a scaled dot-product to compute the *attention weight vector* a_t, which captures the correlation between the *target element* and the *source elements* in the sequence; we define a target element to be the load instruction for which the model will make a prediction, and we define the source elements to be the past load instructions in the sequence that the model uses to make the prediction. Specifically, a_t is computed by first using a scoring function to compare the hidden states of the current target h_t against each source element h_s, before then scaling with a factor f and normalizing the scores to a distribution using the softmax function. The attention weight vector is then used to compute a context vector, c_t, which is concatenated with h_t to determine the output decision. In this work, we use the dot product as the scoring function.

$$a_t(s) = \frac{exp(f \cdot score(h_t, h_s))}{\sum_{s'} exp(f \cdot score(h_t, h_{s'})} \tag{3.3}$$

While the attention's scaling factor was originally used to deal with the growing magnitude of dot products with the input dimension [123], we find a new

use of the scaling factor: A moderate increase in the scaling factor forces sparsity in the attention weight vectors but has minimal influence on accuracy. A sparse attention weight vector indicates that only a few source elements in the sequence influence the prediction. For our caching model, we would expect the attention weights to quantify the correlation between the *target memory access* at time step N and the *source memory accesses* from timesteps 1 to $N-1$. Unfortunately, we find that the attention layer without scaling (or scaling factor of 1) presents a nearly uniform attention weight distribution, thus providing little useful information. To avoid this uniform distribution, we increase the scaling factor f to force the sparsity in attention weight distribution, thereby revealing dependences between source accesses and target access.

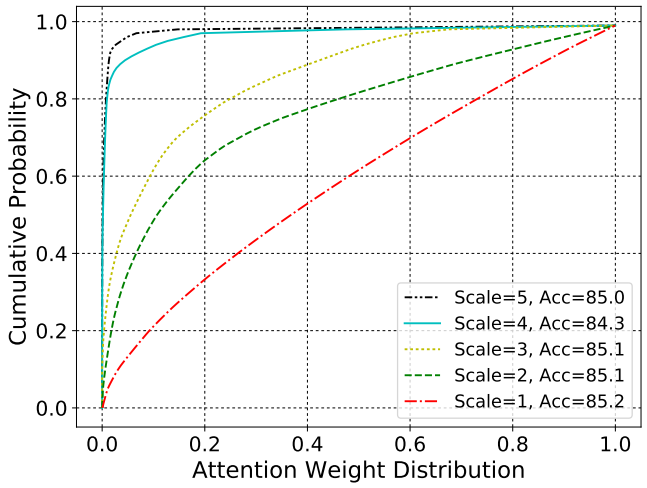


Figure 3.5: Cumulative distribution function of attention weight distribution for omnetpp.

Insights From The Scaled Attention Layer. Figure 3.5 shows for one benchmark the cumulative distribution function of the attention weight distributions with different scaling factors. Surprisingly, we see that without losing accuracy, the scaled attention weight distribution becomes biased towards a few source accesses, which shows that our model can make correct predictions based on just a few source memory accesses.

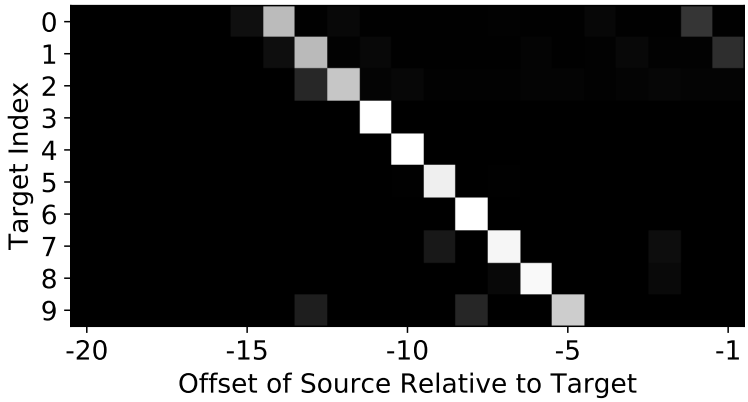


Figure 3.6: Attention weight vectors of consecutive memory accesses. The y-axis shows the indices of target memory accesses, and the x-axis shows the offset of source memory accesses from the target. The white boxes show that target memory accesses are strongly correlated with just a few source memory accesses.

Figures 3.6show the attention weight distributions for 100 and 10 consecutive memory accesses, respectively. Each row represents the attention weight vector for one target memory access; white boxes indicate strong correlation between source and target, and black boxes indicate weak correlation. Figure 3.6 shows that the same source memory access has dominant attention weights for nearly every target, forming an oblique line as its offset increases with the row index. Therefore, we obtain our second observation:

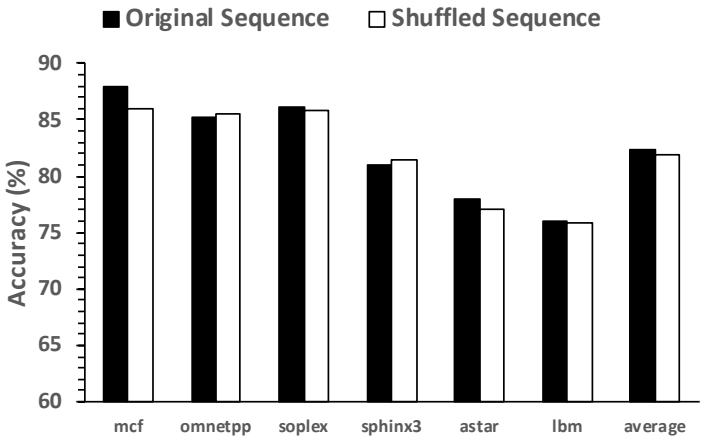


Figure 3.7: Accuracy for the original ordered sequence and the randomly shuffled sequence.

Observation 2. Our model can achieve good accuracy by attending to just a few sources.

From observation 2, we posit that optimal caching decisions depend not on the order of the sequence but on the presence of important PCs. To confirm this conjecture, we randomly shuffle—for time step N—our test sequence from time step 1 to $N - 1$; Figure 3.7 shows that this randomly shuffled sequence sees only marginal performance degradation compared to the original, giving rise to observation 3.

Observation 3. Prediction accuracy is largely insensitive to the order of the sequence.

These observations lead to an important insight into the caching problem.

Important Insight. *Optimal caching decisions can be better predicted with a long history of past load instructions, but they depend primarily on the presence of a few*

PCs in this long history, not the full ordered sequence. Thus, with an appropriate feature design, caching can be simplified from a sequence labeling problem to a binary classification problem.

Sequence 1: PC0, PC1, PC3		**Sequence 2: PC3, PC1, PC0**	
Sequence Representation:	**[1, 0, 0, 0]**	**Sequence Representation:**	**[0, 0, 0, 1]**
	[0, 1, 0, 0]		**[0, 1, 0, 0]**
	[0, 0, 0, 1]		**[1, 0, 0, 0]**
***k*-sparse Binary Feature:**	**[1, 1, 0, 1]**	***k*-sparse Binary Feature:**	**[1, 1, 0, 1]**

Figure 3.8: Examples of k-sparse binary feature. For simplicity, the total number of PCs is 4 and k is 3.

To better understand the program semantics behind this insight, we map source and target PCs back to the source code and find that our model is able to learn high-level application-specific semantics that lead to different caching behaviors of target PCs (see Section 3.3.5).

3.2.3 Integer SVM and a k-sparse Binary Feature

Our insights reveal that it is possible to substitute the LSTM with a simpler model that does not need to capture position and ordering information within the input sequence. Therefore, we simplify our input feature representation to remove duplicate PCs and to forego ordering information, and we feed this simplified hand-crafted feature into a hardware-friendly Integer Support Vector Machine (ISVM). Note that since we remove duplicate PCs, our hand-crafted feature can capture an effective history length of 30 PCs with fewer history elements (5 in our experiments). We denote this compressed history length as k.

In particular, we design a k-sparse binary feature to represent PCs of a

memory access sequence, where a k-sparse binary feature has k 1s in the vector and 0s elsewhere. Specifically, the k-sparse binary feature vector is represented as $\mathbf{x} \in \{0,1\}^u$, where u is the total number of PCs and the tth entry x_t is a $0/1$ indicator, denoting whether the t^{th} PC is within the sequence or not. For a given time step, this vector shows the last k unique PCs for the current memory access. Figure 3.8 shows the one-hot representation and k-sparse binary feature for two sequences. We see that regardless of the order and the position of each PC, the k-sparse representations for two sequences are identical. Thus, our feature design exploits the fact that the order is not important for the optimal caching decision, thereby simplifying the prediction problem.

We then use an SVM with the k-sparse binary feature. Since integer operations are much cheaper in hardware than floating point operations, we use an Integer SVM (ISVM) with an integer margin and learning rate of 1. While several variations exist, we use *hinge loss* as our objective function for optimization, which is defined as

$$l(\mathbf{x}, y) = \max(0, 1 - y \cdot \mathbf{w}^T \mathbf{x}) \tag{3.4}$$

where $\mathbf{w}$ is the weight vector of the SVM and $y \in \{\pm 1\}$ is the expected output.

Fact 1. With binary features, the use of gradient descent with learning rate $\gamma = \frac{1}{n}$ for an integer n is equivalent to optimizing the following objective function with learning rate 1.

$$\tilde{l}(\mathbf{x}, y) = \max(0, n - y \cdot \mathbf{w}^T \mathbf{x}) \tag{3.5}$$

Suppose we are optimizing (3.4) with initial weight vector $\mathbf{w}^{(0)}$, and learning rate $\frac{1}{n}$ produces the trace $\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(m)}, \ldots$. Then optimizing (3.5) with initial weight vector $n \cdot \mathbf{w}^{(0)}$ and learning rate 1 produces $n \cdot \mathbf{w}^{(0)}, n \cdot \mathbf{w}^{(1)}, \ldots, n \cdot \mathbf{w}^{(m)}, \ldots$, which means that they give the same prediction on any training sample. Therefore, by setting the learning rate to one, weight updates will be integral and we can avoid floating point operations. Thus, ISVM trained in an online manner is equivalent to a perceptron [121] that uses a threshold to prevent inadequate training.

ISVM is more amenable to hardware implementation than a vanilla SVM, and because it is a simpler model, it is likely to converge faster than an LSTM model and to achieve good performance in an online manner. In the following experiments, we use $k = 5$. Thus, our Glider solution consists of the ISVM model and k-sparse feature.

3.2.4 Hardware Design

Figure 3.9 shows the hardware implementation of Glider's predictor, which has two main components: (1) a PC History Register (PCHR) and (2) an ISVM Table. The PCHR maintains an unordered list of the last 5 PCs seen by each core; we model the PCHR as a small LRU cache that tracks the 5 most recent PCs. The ISVM Table tracks the weights of each PC's ISVM; we model it as a direct-mapped cache that is indexed by a hash of the current PC and that returns the ISVM's weights for that PC.

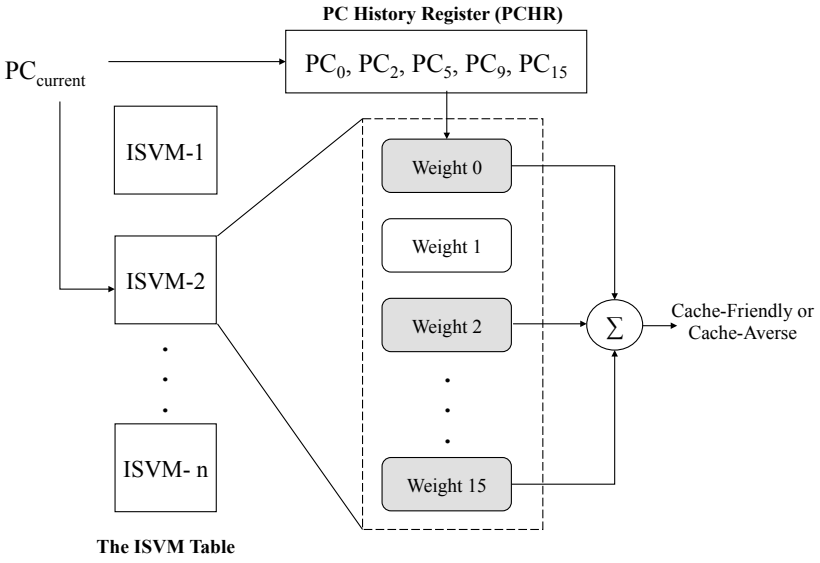


Figure 3.9: The Glider predictor.

Each PC's ISVM consists of 16 weights for different possible PCs in the history register. To find the 5 weights corresponding to the current contents of the PCHR, we create a 4-bit hash for each element in the PCHR (creating 5 indices in the range 0 to 15), and we retrieve the 5 weights at the corresponding indices. For example, in Figure 3.9, the PCHR contains PC_0, PC_2, PC_5, PC_9 and PC_{15}, and we retrieve $weight0$, $weight2$, $weight5$ (not shown), $weight9$ (not shown) and $weight15$ for both training and prediction. A detailed storage and latency analysis is presented in Section 3.3.4.

We now discuss the operations of Glider's predictor in more detail. For details on other aspects of the replacement policy, including insertion and eviction, we refer the reader to the Hawkeye policy [52].

Training. Glider is trained based on the behavior of a few sampled sets [95, 52]. On access to a sampled set, Glider retrieves the weights corresponding to the current PC and the PCHR. The weights are incremented by 1 if OPTgen determines that the line should have been cached; it is decremented otherwise. In keeping with the perceptron update rule [121, 60], the weights are not updated if their sum is above a certain threshold. To find a good threshold, Glider's predictor dynamically selects among a fixed set of thresholds (0, 30, 100, 300, and 3000). While this adaptive threshold provides some benefit for single-core workloads, the performance is largely insensitive to the choice of threshold for multi-core workloads.

Prediction. To make a prediction, the weights corresponding to the current PC and the PCHR are summed. If the summation is greater than or equal to a threshold (60 in our simulations), we predict that the line is cache-friendly and insert it with high priority (RRPV=0[2]). If the summation is less than 0, we predict that the line is cache-averse and insert it with low priority (RRPV=7). For the remaining cases (sum between 0 and 60), we determine that the line is cache-friendly with a low confidence and insert it with medium priority (RRPV=2).

3.3 Evaluation

We evaluate our ideas by comparing in an offline setting our simple ISVM model against a more powerful LSTM model (Section 3.3.2). We then compare

[2]The Re-Reference Prediction Value (RRPV) counter is used by many modern replacement policies [55, 131, 52] and indicates the relative importance of cache lines.

Table 3.2: Baseline configuration.

L1 I-Cache	32 KB, 8-way, 4-cycle latency
L1 D-Cache	32 KB, 8-way, 4-cycle latency
L2 Cache	256 KB, 8-way, 12-cycle latency
LLC per core	2MB, 16-way, 26-cycle latency
DRAM	tRP=tRCD=tCAS=24 800MHz, 3.2 GB/s for single-core, and 12.8 GB/s for 4-core

Glider against other online models, ie, against three of the top policies from the 2^{nd} Cache Replacement Championship (Section 3.3.3), before discussing the practicality of our solution (Section 3.3.4).

3.3.1 Methodology

Simulator. We evaluate our models using the simulation framework released by the 2^{nd} JILP Cache Replacement Championship (CRC2), which is based on ChampSim [1] and models a 4-wide out-of-order processor with an 8-stage pipeline, a 128-entry reorder buffer and a three-level cache hierarchy. Table 3.2 shows the parameters for our simulated memory hierarchy.

Benchmarks. To evaluate our models, we use the 33 memory-sensitive applications of *SPEC CPU2006* [37], *SPEC CPU2017*, and *GAP* [7], which we define as the applications that show more than 1 LLC miss per kilo instructions (MPKI). We run the benchmarks using the reference input set, and as with the CRC2, we use SimPoint to generate for each benchmark a single sample of 1 billion instructions. We warm the cache for 200 million instructions and measure the behavior of the

Table 3.3: Statistics for benchmarks used in offline analysis.

Program	# of Accesses	# of PCs	# of Addrs	Ave. # Accesses per PC	Ave. # Accesses per Addr
mcf	19.9M	650	0.87M	30K	22.9
omnetpp	4.8M	1498	0.44M	3.2K	10.9
soplex	9.4M	2348	0.39M	3.9K	24.1
sphinx	3.0M	1698	0.11M	1.7K	27.3
astar	1.2M	54	0.31M	22K	3.8
lbm	5.0M	55	0.71M	90K	7.0

next 1 billion instructions.

Multi-Core Workloads. Our multi-core experiments simulate four benchmarks running on 4 cores, choosing 100 mixes from all possible workload mixes. For each mix, we simulate the simultaneous execution of the SimPoint samples of the constituent benchmarks until each benchmark has executed at least 250M instructions. If a benchmark finishes early, it is rewound until every other application in the mix has finished running 250M instructions. Thus, all the benchmarks in the mix run simultaneously throughout the sampled execution. Our multi-core simulation methodology is similar to that of CRC2 [1].

To evaluate performance, we report the weighted speedup normalized to LRU for each benchmark mix. This metric is commonly used to evaluate shared caches [52, 60, 1] because it measures the overall performance of the mix and avoids domination by benchmarks of high IPC. The metric is computed as follows. For each program sharing the cache, we compute its IPC in a shared environment (IPC_{shared}) and its IPC when executing in isolation on the same cache (IPC_{single}).

We then compute the weighted IPC of the mix as the sum of $IPC_{shared}/IPC_{single}$ for all benchmarks in the mix, and we normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

Settings for Offline Evaluation. Since LSTM and SVM are typically trained offline—requiring multiple iterations through the entire dataset—we evaluate these models with traces of LLC accesses, which are generated by running applications through ChampSim. For every LLC access, the trace contains a (PC, optimal decision) tuple. The optimal decisions are obtained by running an efficient variant of Belady's algorithm [52]. Because these models require significant training time, we run our offline learning models on 250 millions of instruction for a subset of single-core benchmarks. These benchmarks are statically summarized in Table 3.3. For offline evaluation, we use the first 75% of each trace for training and the last 25% for testing. The models evaluated in this section are insensitive to the split ratio, as long as at least 50% is used for training. For offline evaluation, models are iteratively trained until convergence.

Baseline Replacement Policies. Using the offline settings, we compare the accuracy of the attention-based LSTM and the offline ISVM models to two state-of-the-art hardware caching models, namely, Hawkeye [52] and Perceptron [121].

Hawkeye uses a statistical model that assumes that memory accesses by the same PC have the same caching behavior over a period of time. In particular, Hawkeye uses a table of counters, where each counter is associated with a PC and

is incremented or decremented based on optimal decisions for that PC. **Perceptron** uses a linear perceptron model with a list of features including the PC of the past 3 memory accesses.

The Perceptron model respects the order of these PCs, but we find that using longer PC histories with order is not as effective as without order. For a fair comparison of PC history as the feature, we implement an SVM with the same hinge loss for Perceptron that uses the PC of the past 3 memory accesses, respecting the order, and that learns from Belady's optimal solution.[3]

To evaluate Glider as a practical replacement policy, we compare Glider against **Hawkeye** [52], **SHiP++** [136] and **MPPPB** [60], which are the first, second and fourth finishers in the most recent Cache Replacement Championship (CRC2) [1]. For all techniques, we use code that is publicly available by CRC2. For single-thread benchmarks, we also simulate Belady's optimal replacement policy (MIN) [8].

3.3.2 Comparison of Offline Models

Figure 3.10 compares the accuracy of our models when trained offline. We see that (1) our attention-based LSTM improves accuracy by 10.4% over the Hawkeye baseline and (2) with a 9.1% accuracy improvement over Hawkeye, our offline ISVM comes close to the performance of LSTM. These results confirm our in-

[3] Although our implementation of Percepton has now become quite different from the original implementation in terms of the features, model, and labeling, we still refer to this model as Perceptron in the offline comparison because it is inspired by that work [121].

sight that we can approximate the powerful attention-based LSTM with a simpler hardware-friendly predictor.

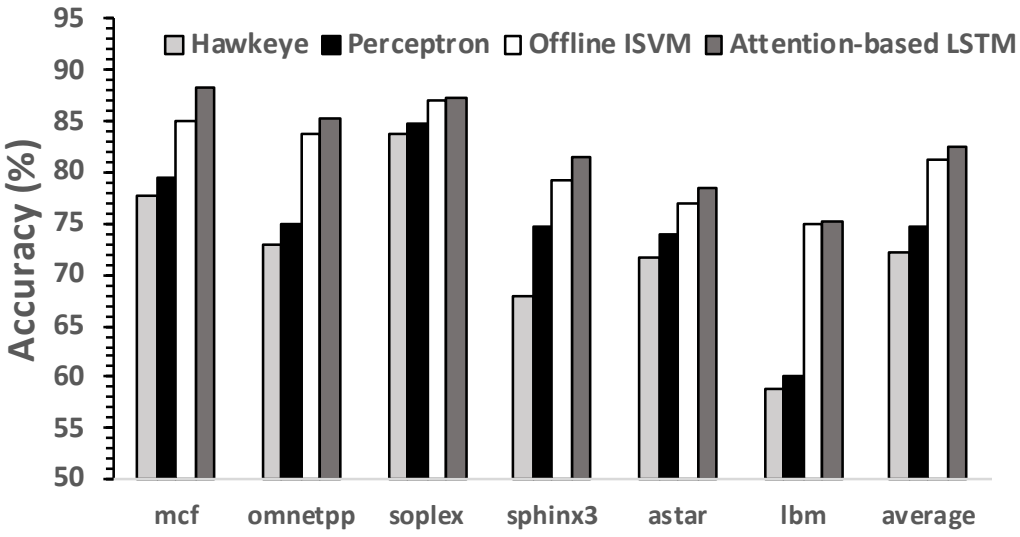


Figure 3.10: Accuracy comparison of offline predictors.

3.3.3 Comparison of Online Models

We now compare the accuracy and speedup of our practical models when trained online as the program executes, i.e., we compare Glider against Hawkeye, SHiP++, and MPPPB.

Online training accuracy. Figure 3.11 shows that Glider is more accurate than state-of-the-art online models, including Hawkeye (88.8% vs. 84.9%). On the subset of benchmarks used for training the offline models, the accuracy improves from 73.5% to 82.4%, which is similar to the offline improvements from 72.2% to 81.2%.

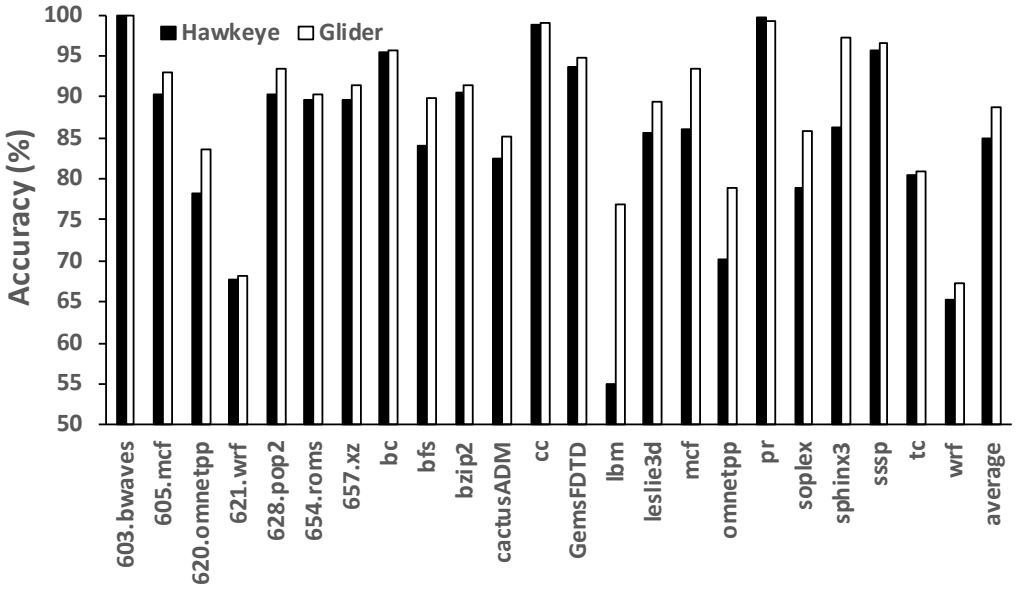


Figure 3.11: Accuracy comparison of online predictors.

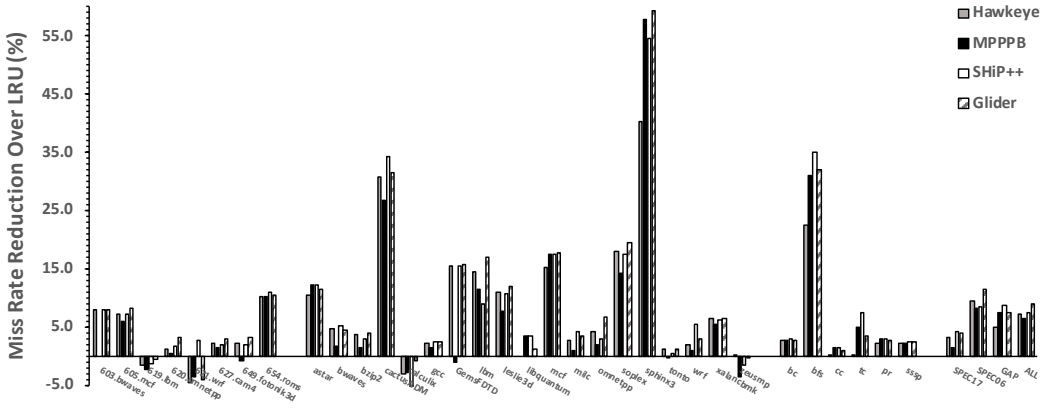


Figure 3.12: Miss rate reduction for single-core benchmarks.

Thus, Glider is as effective as the offline attention-based LSTM model, and insights from offline training carry over to online predictors.

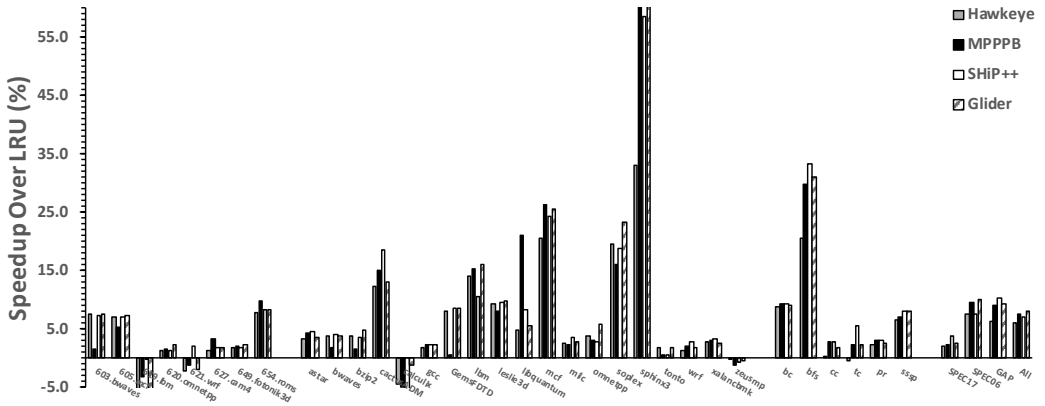


Figure 3.13: Speedup comparison for single-core benchmarks.

Single-Core Performance. Figure 3.12 shows that Glider significantly reduces the LLC miss rate in comparison with the three state-of-the-art replacement policies. In particular, Glider achieves an average miss reduction of 8.9% on the 33 memory-intensive benchmarks, while Hawkeye, MPPPB, and SHiP++ see miss reductions of 7.1%, 6.5%, and 7.5%, respectively. Figure 3.13 shows that Glider achieves a speedup of 8.1% over LRU. By contrast, Hawkeye, MPPPB, and SHiP++ improve performance over LRU by 5.9%, 7.6%, and 7.1%, respectively. These improvements indicate that even though our insights were derived from an offline attention-based LSTM model, they carry over to the design of practical online cache replacement policies.

Multi-Core Performance. Figure 3.14 shows that Glider performs well on a 3-core system as it improves performance by 14.7%, compared with the 13.6%, 11.4%, and 13.2% improvements for Hawkeye, SHiP++, and MPPPB, respectively, indicating that our features and insights are applicable to both private and shared

caches.

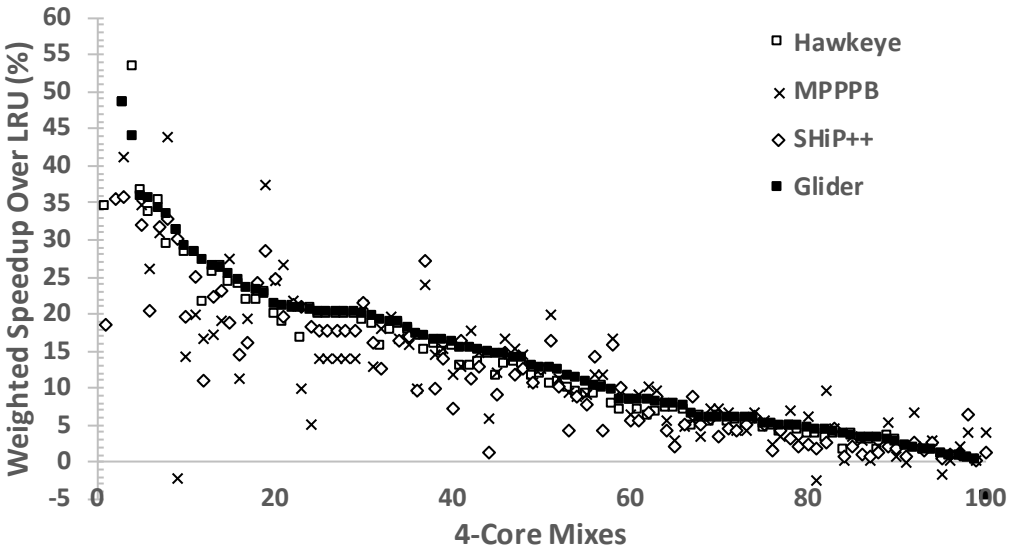


Figure 3.14: Weighted speedup for 4 cores with a shared 8MB LLC.

Effective Sequence Length. Figure 3.15 shows the relationship between history length and offline accuracy, where the sequence length for the attention-based LSTM ranges from 10 to 100, and the number of unique PCs (k value) for offline ISVM and the number of PCs for Perceptron range from 1 to 10. We make three observations. First, the LSTM benefits from a history of 30 PCs, which is significantly larger than the history length of 3 considered by previous solutions [121]. Second, the offline ISVM with only 6 unique PCs approaches the accuracy of the attention-based LSTM; thus, the k-sparse feature representation used by ISVM effectively captures a long history with fewer elements, and this representation works well even with a linear model. Third, the accuracy curve of the perceptron, which

uses an ordered PC history with repetition, does not scale as well as our ISVM, and it saturates at a history length of 4, indicating that the linear model does not work well with an ordered history representation.

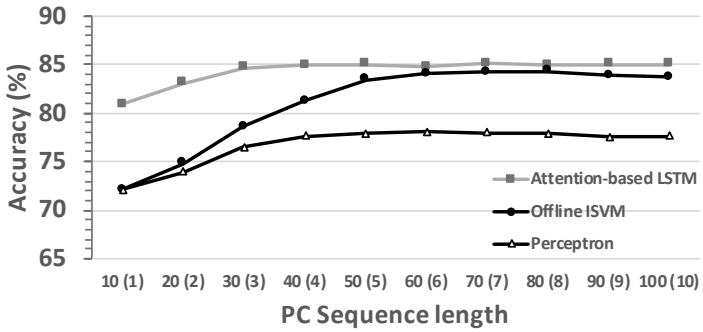


Figure 3.15: Sequence length for attention-based LSTM (number of unique PCs for offline ISVM and sequence length for Perceptron).

3.3.4 Practicality of Glider vs. LSTM

We now compare the practicality of the attention-based LSTM with Glider along two dimensions: (1) hardware budget and (2) training overhead.

Table 3.4: Model size and computation cost. LSTM uses floating point operations; the other models use integer ops.

Model	Model Size (in KB)	Computational Cost per Sample (# operations)	
		Training	Test
LSTM (predictor only)	$\sim 5 \times 10^3$	$\sim 2.4 \times 10^3$	$\sim 0.12 \times 10^3$
Glider	62	8	8
Perceptron	29	9	9
Hawkeye	32	1	1

Hardware Budget of Glider vs. LSTM. In Glider, we replace the predictor module of Hawkeye with ISVM, keeping other modules the same as Hawkeye. For a 16-way 2MB LLC, Hawkeye's budgets for replacement state per line, sampler, and OPTgen are 12KB, 12.7KB, and 4KB, respectively. The main overhead of Glider is the predictor that replaces Hawkeye's per-PC counters with ISVM. For each ISVM, we track 16 weights, and each weight is 8-bit wide. Thus, each ISVM consumes 16 bytes. Since we track 2048 PCs, Glider's predictor consumes a total of 32.8KB. The PCHR with the history of past 5 accesses is only 0.1KB. Thus, Glider's total hardware budget is 61.6 KB. Note that the attention-based LSTM model is at least 3 orders of magnitude more expensive in terms of both storage and computational costs. (See Table 3.4.)

Since the Glider predictor requires only two table lookups to perform both training and prediction, its latency can be easily hidden by the latency of accessing the last-level cache.

Convergence Rate of Glider vs. LSTM. As discussed, deep learning models, such as LSTM, typically need to train for multiple iterations to converge. For caching, training over multiple iterations would imply that a trace of LLC accesses would need to be stored for training iterations, which is infeasibly expensive. Instead, for cache replacement, we need the machine learning model to train in an online manner, that is, by making a single pass over the input data. Figure 3.16 shows that with offline training, our offline ISVM achieves good accuracy in one iteration, while the LSTM takes 10-15 iterations to converge. We also see that on-

line models, such as, Perceptron and Hawkeye, converge fast but have the limited accuracy.

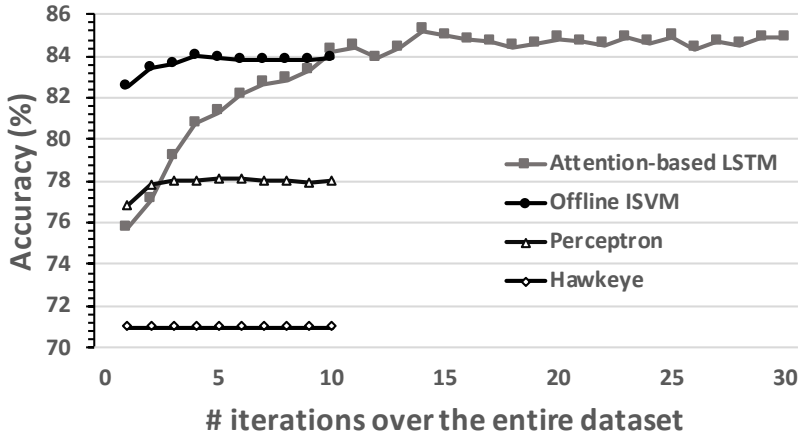


Figure 3.16: Convergence of different models.

On the Practicality of Deep Learning for Caching. The main barriers to the use of deep learning models for hardware prediction are model size, computational cost, and offline training. The model size of our attention-based LSTM is at least 1 megabyte, which significantly exceeds the hardware budget for hardware caches. In addition, LSTM typically requires floating-point operations, while models such as Perceptron and ISVM use integer operations. Fortunately, recent studies [32, 46] have shown the great potential of reducing the model size and computational costs by 30× to 50× through model compression techniques, such as quantization, pruning, and integerization/binarization. However, these models need to be pre-trained offline before being compressed and deployed, which is difficult for hardware prediction problems where program behavior varies from benchmark to benchmark

and even from one input to another input of the same benchmark. Given their problem with underfitting (poor performance in the first 10 iterations) as shown in Figure 3.16, it's clear that even with further compression techniques, deep learning models are still not ready for direct use in hardware predictors.

3.3.5 Learning High-Level Program Semantics

Our attention-based LSTM model is able to learn high-level program semantics to better predict the optimal caching solution. For example, for the *omnetpp* benchmark that simulates network protocols such as HTTP, the model discovers that certain types of network messages tend to be cache-friendly, while other types of messages tend to be cache-averse. Furthermore, the model discovers this relationship by distinguishing the different control-flow paths for different types of messages.

Table 3.5: The attention-based LSTM model improves accuracy for four target PCs in *scheduleAt()* method, and all four target PCs attend to the same source PC.

Target PC	Source PC	Hawkeye's Accuracy(%)	Attention-based LSTM's Accuracy (%)
44c7f6	44e141	74.8	90.1
4600ec	44e141	53.2	94.1
44dd98	44e141	67.1	92.3
43fb10	44e141	73.4	91.0

More specifically, consider the *scheduleAt()* method, which is frequently called inside omnetpp to schedule incoming messages at a given time t. The *scheduleAt()* method takes as an argument a message pointer, and it dereferences this

pointer resulting in a memory access to the object residing at the pointer location (see Figure 3.18). Table 3.5 shows the model's accuracy for four target load instructions (PCs) that access this object. We see that (1) the attention-based LSTM model significantly improves accuracy for all four target PCs, and (2) all four target PCs share the same *anchor PC* (the source PC with the highest attention weight).

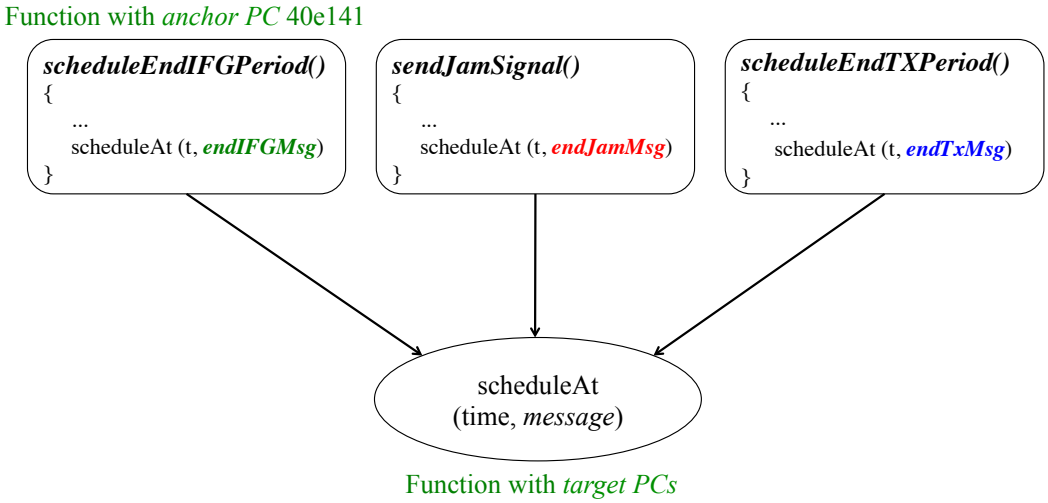


Figure 3.17: The anchor PC belongs to one of the calling contexts for the target PCs.

To understand the accuracy improvement for the target PCs in the *scheduleAt()* method, Figure 3.17 shows that *scheduleAt()* is invoked from various locations in the source code, with each invocation passing it a different message pointer. We find that the anchor PC belongs to one of these calling methods, called *scheduleEndIFGPeriod()*, implying that load instructions in the *scheduleAt()* method tend to be cache-friendly when the *scheduleAt()* method is called from *scheduleEndIFGPeriod()* with the *endIFGMsg* pointer, whereas they tend to be cache-averse when *scheduleAt()* is called from other methods with other message pointers. Thus,

	C code for Source PC 44c7f6
1	int cSimpleModule::scheduleAt(simtime_t t, cMessage *msg)
2	{
3	if (t<simTime())
4	throw new cException(eBACKSCHED);
5	. . .
6	// set message parameters and schedule it
7	**msg->setSentFrom(this, -1, simTime());**
8	msg->setArrival(this, -1, t);
9	ev.messageSent(msg);
10	simulation.msgQueue.insert(msg);
11	return 0;
12	}

	Assembly Code for Target PC 44c7f6		
1	<_ZN13cSimpleModule10scheduleAtEdP8cMessage>:		
2	44c730:	48 89 5c 24 e8	mov %rbx,-0x18(%rsp)
3	. . .		
4	**44c7f6:**	**48 8b 03**	**mov (%rbx),%rax**
5	44c7f9:	48 89 ee	mov %rbp,%rsi
6	44c7fc:	48 89 df	mov %rbx,%rd

Figure 3.18: Source code and assembly code for target PC 44c7f6 in *scheduleAt()* method (lines in bold).

by correlating the control-flow histories of load instructions in the *scheduleAt()* method, our model has discovered that the *endIFGMsg* object has better cache locality than *endJamSignal* and *endTxMsg* objects.

3.3.6 Model Specifications

The hyper-parameters for the attention-based LSTM model and Glider are given in Table 3.6. Here we explain how we identify key hyper-parameters, namely, the sequence length for the attention-based LSTM model and the number of unique PCs (k) for Glider and Perceptron.

For the offline ISVM, we consider step sizes n from 0.0001 to 1 with a multiple of 5 (0.0001, 0.0005, 0.001, 0.005, ...), and for the corresponding Glider model we use an update threshold of $\frac{1}{n}$ with a fixed step size of 1. To avoid the need to perform floating point operations, no decay is used.

Table 3.6: Offline Model Specifications

LSTM	train/test split	0.75/0.25
	embedding size	128
	network size	128
	Optimizer	Adam
	learning rate	0.001
Glider	k (# unique PCs)	5
	step size	0.001

3.4 Summary

When we think of applying machine learning to hardware prediction problems, we often focus on the learning model. For example, as a learning model, the perceptron has benefits over tables of saturating counters because the perceptron can efficiently combine results from multiple inputs. And indeed, this work has presented a new cache replacement policy that uses an SVM—which is equivalent

to a perceptron—to outperform the state-of-the-art. However, the key to Glider's success is not the mere use of a perceptron—we are not the first to use perceptrons for cache replacement. Instead, it is the use of appropriate features in combination with the perceptron that leads to its effectiveness. Thus, this work has shown how we can use deep learning—in an offline setting—to derive insights that lead to an improved set of features with which to make predictions for cache replacement.

More broadly, our approach in designing Glider suggests that deep learning can play an important role in systematically exploring features and feature representations that can improve the effectiveness of much simpler models, such as perceptrons and SVMs. However, domain knowledge is critical for this approach to work. In particular, the domain expert must formulate the problem appropriately, supply relevant features to build an effective offline model, and use indirect methods to interpret the trained model. Our work illustrates one instance where this approach is successful, but we hope that the insights and techniques presented in this work can inspire the design of similar solutions for other microarchitectural prediction problems, such as branch prediction, data prefetching, and value prediction.

Chapter 4

Voyager Data Prefetcher

4.1 Challenges of Data Prefetching as a Machine Learning Problem

Unlike cache replacement, data prefetching presents two challenges to machine learning that branch prediction and cache replacement do not.

First, data prefetching suffers from the *class explosion* problem. While cache replacement can be casted as a binary prediction problem [60, 67, 52, 107, 130], prefetchers that learn delta correlations or address correlations have enormous input and output spaces. For a 64-bit address space, the model needs to predict from among tens of millions of unique address values, a feature space that cannot be handled by existing machine learning models for natural language, which traditionally have input and output spaces that are orders of magnitude smaller.

Second, data prefetching has a labeling problem. Whereas branch predictors can be trained by the ground truth answers as revealed by a program's execution, and whereas cache replacement policies can be trained by learning from Belady's MIN policy [52], data prefetchers have no known optimal solution from which to learn. Thus, given a memory access m, the prefetcher could learn to prefetch any of the many addresses that follow m.

4.2 Problem Formulation

As a foundation for our ML solution, we first derive a probabilistic formulation for data prefetching that (1) outlines the design space for data prefetchers, illustrating that features and labels can be used to describe a wide range of existing data prefetchers within a unified framework and (2) motivates the use of ML models to estimate the probability distribution.

4.2.1 Probabilistic Formulation of Prefetching

The goal of temporal prefetching is to exploit correlation between consecutive addresses to predict the next address. Therefore, temporal prefetching can be viewed as a classification problem where each address is a *class*, and the learning task can be defined as the probability that an address $Addr$ will be accessed given a history of past events, such as the occurrence of memory accesses $Access_1, Access_2, ..., Access_t$ up to the current timestamp t:

$$P(Addr|Access_1, Access_2, ..., Access_t) \tag{4.1}$$

Machine learning, especially deep learning, provides a flexible framework for modeling probability distributions. In ML terminology, the historical events ($Access_1$, $Access_2$ ..., $Access_t$) are known as *input features*, and the future event ($Addr$) is known as the model's *output label*.

All previous temporal prefetchers [126, 51, 5] can be viewed as instances of this formulation with different input features and output labels. For example,

STMS [126] learns the temporal correlation between consecutive addresses in the global memory access stream, so its output label is the next address in the global memory access stream. Thus, STMS tries to learn the following probability distribution:

$$P(Addr_{t+1}|Addr_t) \tag{4.2}$$

ISB [51] implements PC localization, which improves upon STMS by providing a different output label, namely, the next address by the same program counter (PC). Thus, ISB tries to learn the following probability distribution:

$$P(Addr_{PC}|Addr_t) \tag{4.3}$$

where $Addr_{PC}$ is the next address that will be accessed by the PC that just accessed $Addr_t$.

Domino [5] instead improves upon STMS by using a different input feature, using the previous two addresses to predict the next address in the global memory access stream:

$$P(Addr_{t+1}|Addr_{t-1}, Addr_t) \tag{4.4}$$

Stride prefetchers can also be formulated under this probabilistic framework by incorporating strides or deltas in our formulation. For example, a stride prefetcher detects the constant stride pattern:

$$P(Stride_{t+1}|Stride_t) \tag{4.5}$$

The VLDP prefetcher [106] looks at a history of past deltas and selects the most likely deltas.

$$P(Stride_{t+1}|Stride_{t_0}, Stride_{t_1}, ..., Stride_{t_n}) \tag{4.6}$$

The first neural prefetcher [33] adopts a similar formulation. Given a history length l, it learns the following distribution:

$$P(Stride_{t+1}|Stride_{t-l}, Stride_{t-l+1}, ..., Stride_t) \tag{4.7}$$

In general, our probabilisitc formulation of prefetching defines the input features (the historical event) and the output label (the future event). Unlike previous learning-based work that focuses on limited features (without addresses) and the prediction of the global stream, we explore the design choices of both the input features and output labels to improve the predictive accuracy of our model.

4.3 Our Solution: Voyager

This section describes Voyager, our neural model for performing data prefetching, including a discussion of the challenges that we had to overcome in developing such a solution.

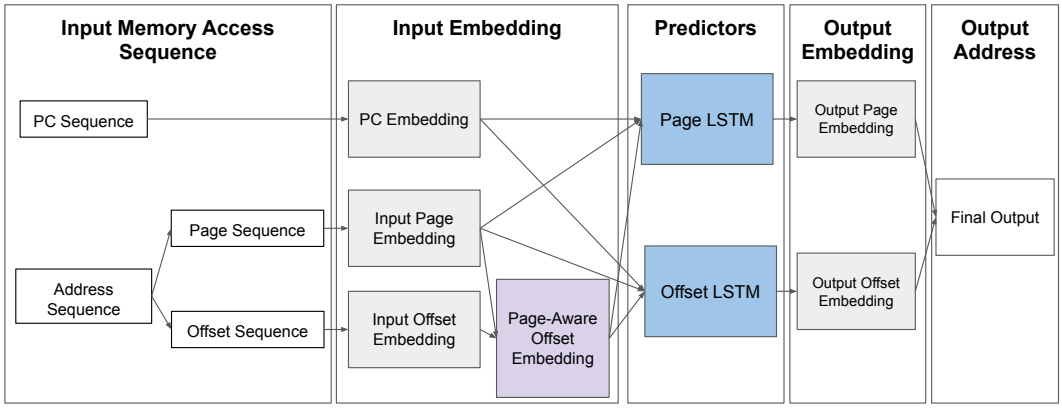


Figure 4.1: Overview of Voyager.

4.3.1 Model Overview

Figure 4.1 shows an overview of Voyager. The model takes as input a sequence of memory accesses, where each memory access is represented by a PC and an address. Our model is hierarchical, so one part of the model predicts page addresses (bits 11-63) and the other predicts offsets within a page (bits 6-11). Since our inputs are categorical in nature—their numerical values are not pertinent to the prediction task—our model first learns embeddings for PC, page, and offset, and then feeds them to an LSTM neural network. The neural network generates predictions of the output page and offset, which are combined to generate a final address prediction. We now discuss our hierarchical neural structure and multi-label training scheme in more detail.

4.3.2 Hierarchical Neural Structure

To better understand the need for a hierarchical model, Table 1 shows that the number of unique addresses in typical programs ranges from hundreds of thou-

Table 4.1: Benchmark statistics.

Benchmark	# PCs	# Addresses	# Pages
astar	192	0.15M	29.9K
bfs	828	0.16M	4.1K
cc	529	0.26M	4.3K
mcf	169	4.58M	91.1K
omnetpp	1101	0.48M	36.3K
pr	650	0.27M	4.2K
soplex	2129	0.36M	12.3K
sphinx	1519	0.13M	4.3K
xalancbmk	2071	0.34M	25.3K
search	6729	0.91M	22.4K
ads	21159	1.4M	28.7K

sands to tens of millions, which is orders of magnitude larger than the number of unique categories in traditional ML tasks, such as words in natural language processing. As pointed out by previous work [33, 107], the explosion of memory addresses leads to an increase in memory usage that precludes the training of neural networks and that results in an ineffective representation of the memory address space, since address only appears a few times, which is insufficient for training neural networks. Therefore, to make prefetching a tractable ML problem, we need to design an efficient representation of memory address space with less unique classes.

Fortunately, the number of unique pages is 30-60$\times$ smaller than the number of unique addresses, so Voyager leverages this property to reduce the number of classes that the ML model needs to learn. As a naive realization of this idea, at each step of the memory address sequence, the input representation is the concatenation of the page embedding, offset embedding and pc embedding, which is fed to the page LSTM and offset LSTM for the prediction of the page and offset of the

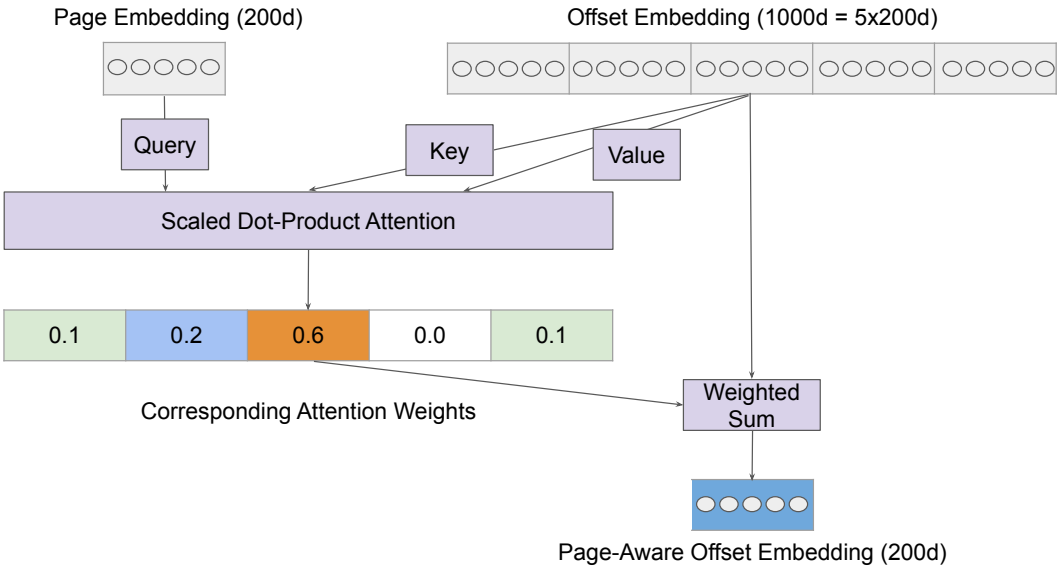


Figure 4.2: Page-aware offset embedding with the dot-product attention mechanism.

future address. The outputs from both LSTMs are taken by a linear layer with softmax activation function (not shown in Figure 4.1), which outputs the probability distribution of pages and offsets respectively.

Unfortunately, naively splitting addresses into page and offsets results in a problem, which we refer to as the *offset aliasing issue*. To understand this aliasing issue, consider two addresses X and Y that have different page numbers P_X and P_Y but the same offset O. Even though X and Y have the same page offsets, their program semantics could differ, and there is no reason to believe that the two will always behave the same way, so it is important for the ML model to distinguish between the same offset of these two addresses. The analogy in natural language is polysemy where multiple meanings exist for a word, and the actual meaning depends on the context in which the word is used; without using this context, the

models learn an average behavior, which is not useful.

4.3.3 Page-Aware Offset Embedding Mechanism

Thus, to effectively represent an address, its an ideal offset embedding not only represents the offset but also includes the impact of the page that it resides on. To achieve this, we propose a novel *page-aware offset embedding mechanism* that makes the offset embedding aware of the page number that the offset originated from. Figure 4.2 illustrates this page-aware offset embedding mechanism.

To build a page-aware offset embedding, we take inspiration from mixtures of experts [49] where each expert represents a specific page-aware offset embedding. Conditioned on the page, our model selects the appropriate expert and outputs its offset embedding. We use the scaled dot-product attention layer [123] as the core mechanism, as it represents the building block of state-of-the-art NLP models [97].

The attention layer can be thought of as a memory lookup, where each element of memory is addressed by a key and stores a vector value. Given a query (the page embedding), the mechanism compares its correlation with each key and outputs a probability vector. The final output value is therefore a sum over the values in the memory, weighted by these probabilities. This mechanism is known as soft attention and allows us to use backpropagation to learn these vectors. We use the offset embeddings for each expert to represent both the keys and values.

Formally, we can think of the offset embedding as one large vector, and we can think of each expert as being partitions of this vector (see Figure 4.2). When we set the ratio between page embedding size and total offset embedding size to be n,

corresponding to n experts, the mechanism can be defined as

$$a_t(o,s) = \frac{exp(f \cdot score(h_p, h_{o,s}))}{\sum_{s'} exp(f \cdot score(h_p, h_{o,s'})} \tag{4.8}$$

$$h'_o = \sum_s a_t(o,s) h_{o,s} \tag{4.9}$$

where f is a scaling factor that ranges from 0 to 1, h_p is the page embedding, $h_o = [h_{o,0}, h_{o,1}, ..., h_{o,n}]$ is the offset embedding, where $h_{o,i}$ is the embedding of the i^{th} expert, and h'_o is the page-aware offset embedding generated by the attention mechanism. Empirically, we set the size of the offset embedding $|h_o|$ to be 5-100$\times$ of that of the page embedding $|h_p|$. In the example in Figure 4.2, we use a dot-product attention layer with a 200-dimension (d) page embedding ($|h_p| = 200$) and 1000-dimension (d) offset embedding ($|h_o| = 1000$). The 1000-d offset embedding $|h_o|$ is divided into 5 expert embeddings ($n = 5$), each of which is the same size as the page embedding used to perform the attention operation. Attention weights $a_t(o,s)$ are computed as the dot product of the page embedding and each of the offset expert embeddings, and a final page-aware offset embedding h'_o is obtained by a weighted sum of all the offset expert embeddings $h_{o,k}, k = 0, 1, ..., n$.

As a result of the page-aware offset embedding, the input representation of Voyager's LSTM model becomes the concatenation of the page embedding, page-aware offset embedding and pc embedding. The hierarchical structure assisted by the page-aware offset embedding mechanism significantly reduces the number of classes in both the input and output space because the number of offsets is fixed

(64) and is much smaller than the number of pages for all programs. For networks with a large number of classes, the embedding layer is the primary storage and computation bottleneck. Reducing the number of classes dramatically reduces the size of the model by decreasing the number of embedding entries and parameters. This reduction consequently simplifies the model and reduces training overhead. In Section 4.4 we show that Voyager improves the model efficiency—in terms of computational cost and storage overhead—by an order of magnitude when compared to previous neural-based solutions [33].

Covering Compulsory Misses with Deltas. Compulsory misses are common in benchmarks with large memory footprints, such as *mcf* and *search*. To improve coverage without burdening the model with infrequent addresses, we train Voyager to also learn deltas among addresses, allowing it to predict compulsory misses. In particular, we focus on addresses with extremely low frequency (e.g. fewer than 2 times) and we represent these addresses using their deltas from previous addresses. By focusing on addresses with low frequency, our model can use just a small number of deltas. For example, we find that 10 deltas can cover 99% of the compulsory misses in *mcf*, whereas previous solutions [33] need millions of deltas.

4.3.4 Multi-Label Training Scheme

As explained in Section 1, data prefetchers do not have access to obvious ground truth labels. We find that different labeling schemes work well for different workloads: Spatial labeling schemes work well for workloads that have spa-

tial memory access patterns, and PC-based labeling schemes work well on pointer-based workloads. In fact, some workloads have a mix of access patterns that require multiple labeling schemes. Thus, we train Voyager with multiple labels so that it can leverage the benefits of different labeling schemes and so that it can select the most predictable label during prediction. To evaluate the effectiveness of previously proposed labeling schemes, we also show results in which we train the model with a single label. Section 4.4.3.3 discusses and evaluates these labeling schemes.

4.4 Evaluation

This section evaluates our ideas by comparing Voyager against both practical prefetchers and neural prefetchers.

4.4.1 Methodology

Due to slow training times, machine learning solutions—including previous machine learning based prefetchers—are typically evaluated in an offline setting, which means that they are trained on one portion of the benchmark's execution and then tested on a different portion of the benchmark's execution. Of course, hardware prefetchers are trained as the program run, so to simulate such an online deployment, we evaluate Voyager (and the baseline machine learning-based prefetchers) by training them in batches of 50 million instructions, so that the prefetchers are trained for one 50M instruction epoch and then used for the next 50M instruction epoch.

Simulator. We evaluate our models using the simulation framework released by the 2nd JILP Cache Replacement Championship (CRC2), which is based on ChampSim [54]. ChampSim models a 4-wide out-of-order processor with an 8-stage pipeline, a 128-entry reorder buffer and a three-level cache hierarchy. Table 4.2 shows the parameters for our simulated memory hierarchy.

Table 4.2: Simulation configuration.

L1 I-Cache	64 KB, 4-way, 3-cycle latency
L1 D-Cache	64 KB, 4-way, 3-cycle latency
L2 Cache	512 KB, 8-way, 11-cycle latency
LLC per core	2MB, 16-way, 20-cycle latency
DRAM	tRP=tRCD=tCAS=20 2 channels, 8 ranks, 8 banks 32K rows, 8GB/s bandwidth per core

Benchmarks. We evaluate Voyager and the baselines on a set of irregular benchmarks from the SPEC06 and GAP benchmark suites [7]. In particular, we use irregular benchmarks on which an oracle prefetcher that always correctly prefetches the next load produces ¿10% IPC improvements over the baseline with no prefetching. This results in a similar subset used by previous work [51, 134, 133]. For each benchmark, we generate traces of length 250 million instructions, and the desired region of execution is identified using the SimPoint [31] methodology. We use the reference input set for SPEC06 and input graphs of size 2^{17} nodes for GAP.

For more challenging workloads, we also evaluate our solution on Google's *search* and *ads*, which are state-of-the-art enterprise-scale applications.

Baseline Prefetchers. We compare Voyager against spatial prefetchers (the Best Offset Prefetcher [83]), temporal prefetchers (STMS [126], ISB [51] and Domino [5]), and impractical neural prefetchers (Delta-LSTM [33]). Since our goal is to evaluate the prediction capabilities of different solutions, we use idealized implementations of all baselines, so there are no constraints on model storage or off-chip metadata, and all storage is accessed with no cost. Our baselines are particularly optimistic for the temporal prefetchers, which typically require 10-100M of off-chip metadata, so practical implementations would incur the latency and traffic overhead of accessing this off-chip metadata.

Metrics. We evaluate our solutions by comparing their accuracy, coverage and IPC over a system with no prefetcher, along with a combined notion of accuracy/-coverage. For effective training, neural models need to be given a clear objective function, but prefetching often represents a tradeoff between coverage and accuracy. Thus, we follow Srivastava et al. [117] and use a strict definition of accuracy/-coverage that unifies accuracy and coverage into a single metric: *We consider the model's prediction to be correct if it correctly predicts the next load address.* This metric unifies accuracy and coverage because each correct prediction improves both accuracy (as it is correct) and coverage (as the next address is covered). From a prediction perspective, this unified metric means that Voyager is designed to improve both accuracy and coverage simultaneously. We also report the conventional separate coverage and accuracy numbers that are obtained from the actual simulation.

In the evaluation of prefetchers with higher degrees (Section 4.4.3.1), the

main goals are to see if the baseline prefetchers with aggressive degree and unlimited resources could capture the patterns covered by Voyager. For these evaluations, we use the standard definition of coverage, which is the percentage of memory accesses that are correctly prefetched.

We also compare the overhead of Voyager, including computational cost and model size, against both a non-hierarchical neural network implementation [33] and a temporal prefetcher [51].

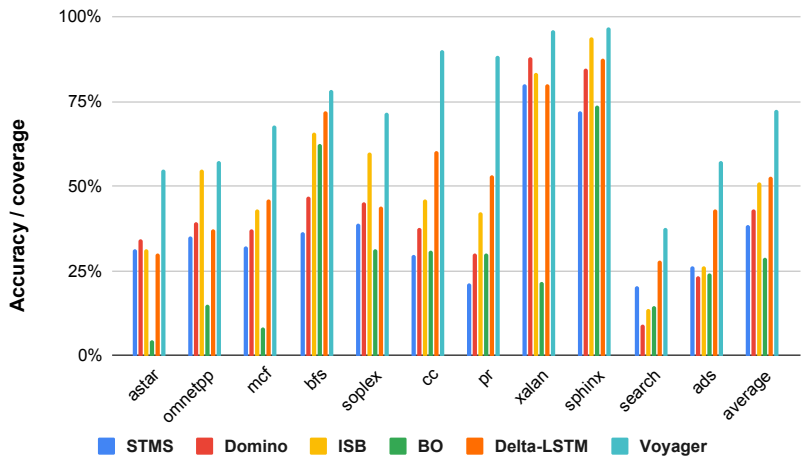


Figure 4.3: Unified accuracy/coverage, including Google's Search and Ads.

4.4.2 Comparison With Prior Art

Figure 4.3 shows that Voyager achieves superior accuracy/coverage compared to the baselines. On average, Voyager achieves 73.9% accuracy/coverage, compared with 38.6% for STMS, 43.3% for Domino, 51.1% for ISB, 20.5% for IP-stride, 28.8% for BO, and 52.9% for the Delta-LSTM. A closer look at Figure 4.3

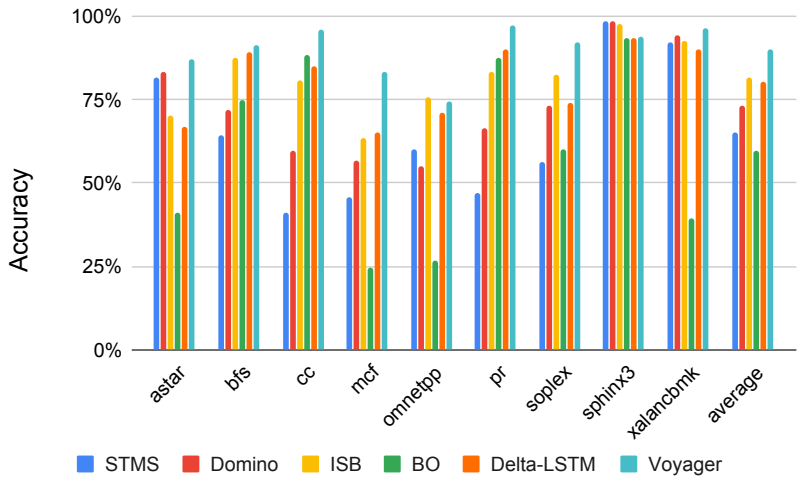


Figure 4.4: Accuracy.

shows that Voyager is particularly effective for Google's *search* and *ads* where it improves accuracy/coverage to 37.8% and 57.5% compared to 27.9% and 43.1% by Delta-LSTM.

While the single accuracy/coverage metric is helpful for evaluating the neural model, we also present coverage and accuracy from simulation. Figure 4.4 and 4.5 show that Voyager improves the accuracy of our SPEC and GAP benchmarks from 81.6% to 90.2% and coverage from 47.2% to 65.7%.

Figure 4.6 shows that Voyager achieves a higher IPC improvement than prior art. Normalized to a baseline that has no prefetcher, Voyager improves performance by 41.6%, compared with 14.9% for STMS, 21.7% for Domino, 28.2% for ISB, 6.9% for IP-stride, 13.3% for BO, and 24.6% for Delta-LSTM.

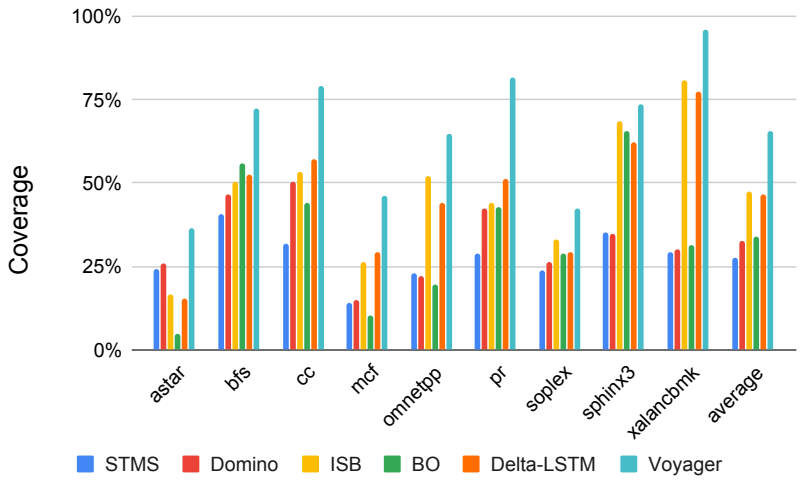


Figure 4.5: Coverage.

4.4.3 Understanding Voyager's Benefits

In this section, we provide an in-depth analysis of Voyager by (1) comparing Voyager against baselines with higher prefetch degrees and against baselines that hybridize prefetchers, (2) analyzing the memory access patterns which account for Voyager's improved coverage, and (3) studying the effectiveness of different features and labels.

4.4.3.1 Prefetch with High Degree

For these evaluations, we use a standard definition of coverage, which is the percentage of memory accesses that are correctly prefetched by each prefetcher.

Figure 4.7 shows that as we increase degree from 1 to 8, the coverage of all prefetchers grows, but even with a degree of 8, a hybrid of ISB+BO can barely

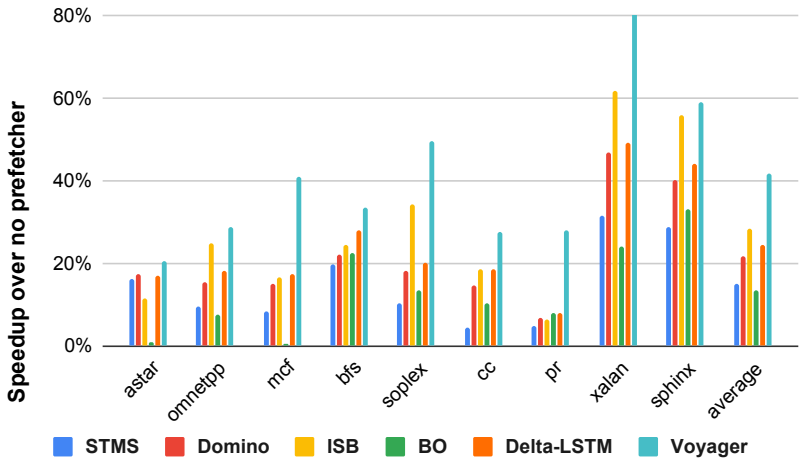


Figure 4.6: IPC.

reach the coverage of Voyager with a degree of 1. Note that in the hybrid prefetcher, ISB, and BO equally share the available degree if higher than 1, and with a degree of 1, the hybrid falls back to ISB.

4.4.3.2 Access Patterns Breakdown

To better understand Voyager's benefits for temporal prefetching, we create a version of Voyager that is trained using only addresses and without deltas. We call this version Voyager w/o delta, and much like ISB, Voyager w/o delta cannot prefetch any compulsory misses. We find that Voyager w/o delta achieves 19.4% better coverage than ISB, indicating Voyager's superiority in predicting temporal access patterns.

We further classify the coverage into spatial and non-spatial patterns; a prefetch candidate is considered to be spatial if the distance between the last address

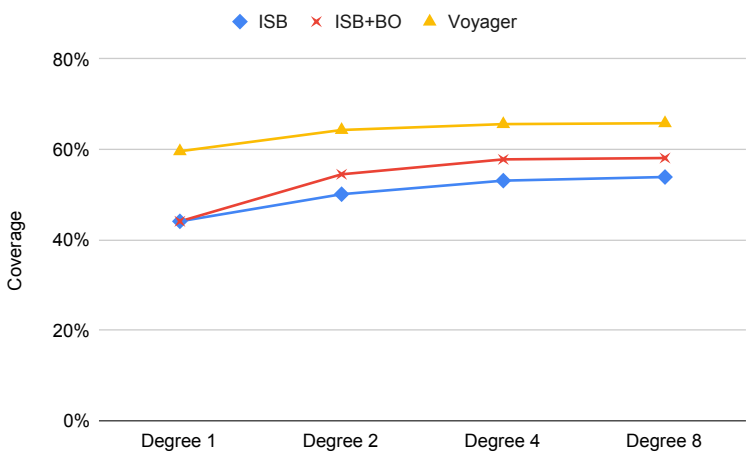


Figure 4.7: Sensitivity to Prefetch degree.

and the prefetched address is less than a certain threshold (256 cache lines [83]). Figures 4.8 and 4.9 show that Voyager w/o delta improves the prediction of spatial patterns from 45.2% to 56.8%, and it improves the prediction of non-spatial patterns from 13.1% to 22.2%.

Finally, to understand uncovered cases, we further classify the uncovered patterns of Voyager w/o delta and ISB into several categories: (1) *uncovered spatial* refers to spatial patterns that are not covered, (2) *uncovered co-occurrence-k* refers to non-spatial patterns that are commonly seen patterns with the top-k frequency, (3) *uncovered others* refers to remaining non-spatial patterns that are not seen too often, and (4) *uncovered compulsory* refers to compulsory misses, which are addresses that have not been seen in the training set. Not surprisingly, we see that Voyager w/o delta reduces the percentage of all types of uncovered patterns except compulsory misses.

Of course, compulsory misses are important for benchmarks with large

memory footprints, such as *mcf*, *search* and *ads*. Since machine learning frameworks are flexible, Voyager can easily include the 10 most frequent deltas as labels. On *mcf*, this reduces the ratio of compulsory misses from 21.6% to 0.2%, improving total coverage from 49.1% to 68%.

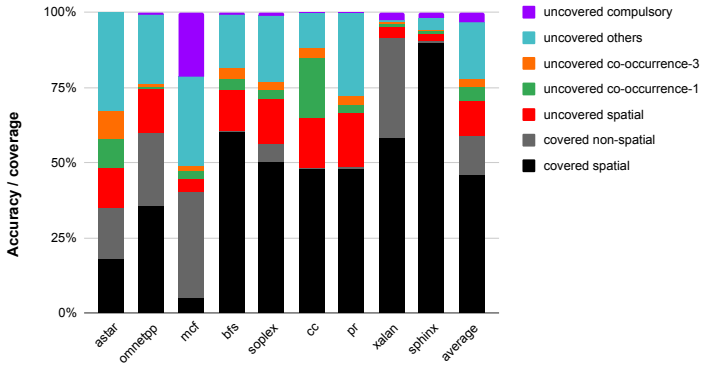


Figure 4.8: Breakdown of the patterns of ISB.

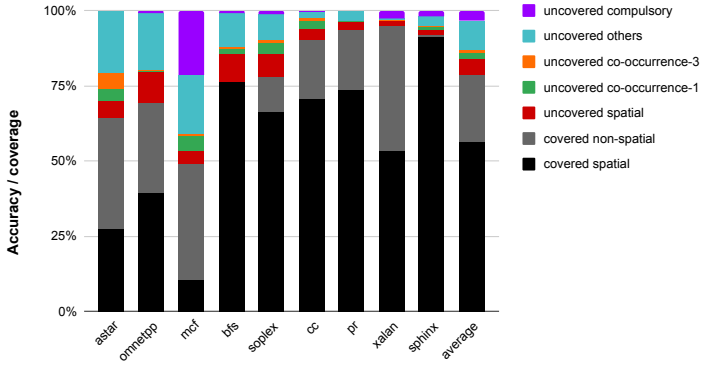


Figure 4.9: Breakdown of the patterns of Voyager w/o delta.

4.4.3.3 Features and Labels

Voyager improves accuracy/coverage by introducing both new features and

a new labeling scheme. This section isolates the effects of each change to understand their relative benefits.

Features. By effectively representing the memory address space with a hierarchical structure, Voyager is able to leverage neural networks to utilize data addresses as features. We study the effectiveness of Voyager's new features by fixing the labeling scheme. In particular, to evaluate the effectiveness of the memory address sequence as a feature, we compare STMS against a version of Voyager that uses the next address in the global stream as the label, which we refer to as Voyager-global. We also compare ISB against a version of Voyager that uses the next address of the current PC as the label; we refer to this version as Voyager-PC.

Figure 4.10 shows that Voyager-global improves coverage over STMS by 19.8%, and Voyager-PC improves coverage over ISB by 16.4%. The right two bars represent two versions of Voyager-PC, one that uses the PC history as a feature and one that does not. We see that unlike with in branch prediction [56, 120, 81] and cache replacement [107], control flow does not help prefetching. Thus, we conclude that for prefetching, PC is not a useful *feature*. However, as we will see shortly, PC *is* useful for labeling.

Labeling. We now evaluate six different labeling schemes: (1) **global** predicts the next address in the global stream, (2) **PC** predicts the next address by the same PC, (3) **basic block** predicts the next address by the PCs from the current basic block, (4) **spatial** predicts the next address within a spatial offset of 256 [83], (5)

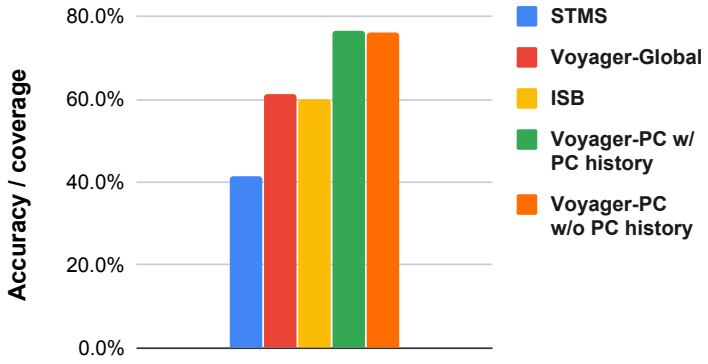


Figure 4.10: Comparison of different features.

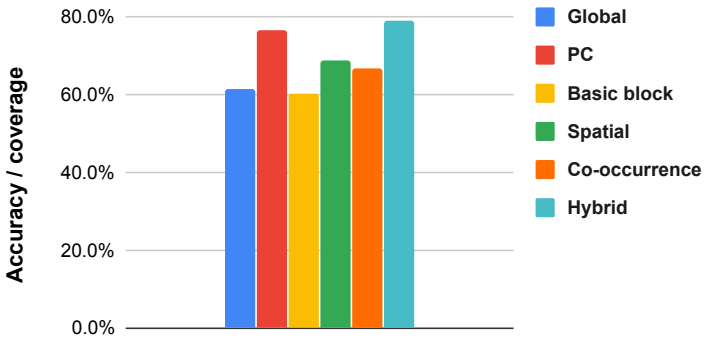


Figure 4.11: Comparison of different labeling schemes.

co-occurrence predicts the address that occurs most often in the future window of 10 memory accesses and (6) **hybrid** combines multiple labeling schemes, including PC, spatial and co-occurrence.

Figure 4.11 compares the accuracy/coverage of these different labeling schemes, and we see that the hybrid scheme is the most effective, but the standalone PC-based scheme performs well too. We also see that basic block labeling is not better than global labeling. This is not surprising because a basic block consists of multiple PCs, so the basic block scheme is the same as the global scheme except for the last load in the basic block.

4.4.4 Why ML-Based Prefetchers

To understand why machine learning models, such as LSTM, are beneficial for prefetching, we provide intuitive analysis by providing code examples to understand how Voyager interprets program semantics through its features and labels.

Features. Figures 4.12 and 4.13 demonstrate the benefit of utilizing data address history as a feature. The code is from the Gap benchmark PageRank, which takes graph-structured inputs. Two loads appear in lines 44 and 49. The load in line 44 is easy to predict since it simply traverses all nodes, or ABCD in the example input graph. Line 49 is more complex, as it traverses all neighbors of all nodes where each node can be a neighbor of many other nodes. Thus, the next node to be accessed depends on both the current neighbor node and the parent node (in bold) of the current neighbor. The prediction of the next access becomes harder since the notion of

line	Code	Prefetch Accuracy (Baseline -> Voyager)
43	`for (NodeID n=0; n < g.num_nodes(); n++)`	
44	`outgoing_contrib[n] = scores[n] / g.out_degree(n);`	99.5% -> 99.5%
45	`for (NodeID u=0; u < g.num_nodes(); u++) {`	
46	`ScoreT incoming_total = 0;`	
47	`for (NodeID v : g.in_neigh(u))`	
48	`incoming_total += outgoing_contrib[v];`	23.5% -> 95.1%
49	`ScoreT old_score = scores[u];`	
50	`scores[u] = base_score + kDamp * incoming_total;`	
51	`error += fabs(scores[u] - old_score);`	

Figure 4.12: Code example from PageRank.

a parent node does not exist from the hardware perspective. For example, depending on the parent node, node B can be followed by any other node, which confuses existing temporal prefetchers that only look at one or two past data addresses. Voyager, however, accurately prefetches line 49's load, since it is able to recognize the parent node through the sequence of unique neighbor nodes. For example, based on the neighbors A and C, Voyager can recognize that the parent node is A and predict that the next neighbor is D.

Labeling. The second code example, shown in Figure 4.14, illustrates the importance of labeling schemes in predicting memory access patterns. The code comes from *soplex* in the SPEC06 benchmark suite. Lines 98-120 determine the value of the variable *leave*. This variable is used later, if greater than 0, to index the arrays *upd*, *ub*, *lb* and *vec*. Voyager prefetches the load of *upd*, *ub* and *lb* by learning from

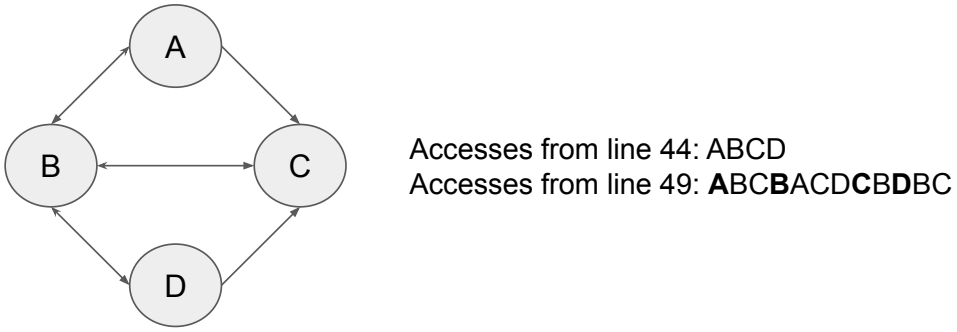


Figure 4.13: An example input graph to PageRank.

the data address sequence with PC localization. One particularly interesting pattern corresponds to *vec* in lines 125 and 127. *vec[leave]* will be accessed regardless of whether the branch is actually taken, but it will be accessed by one of the two PCs (line 125 or 127), depending on the outcome of the branch. From the perspective of either individual PC, the access to *vec* is hard to predict, as the pattern is shared across the two different PCs. However, our co-occurrence labeling scheme correlates *vec[leave]* with *upd[leave]*, since it is always accessed after *upd[leave]*. This correlation makes the pattern more predictable, so Voyager significantly improves over the baseline by prefetching *vec[leave]* at the point of *upd[leave]*.

4.4.5 Model Compression and Overhead

Voyager's hierarchical representation yields significant storage and computational efficiency, when compared against Hashemi et al.'s Delta-LSTM prefetcher [33]. In particular, Voyager improves the training overhead by 15.1 $\times$ and prediction latency by 15.6$\times$. Although the prediction is still relatively slow (18000 nanoseconds per prediction), this is mainly because Tensorflow's Python

line	Code	Prefetch Accuracy (Baseline -> Voyager)
121	`if (leave >= 0)`	
122	`{`	
123	`x = upd[leave];`	upd: 62.9% -> 94.1%
124	`if (x < epsilon)`	
125	`val = (ub[leave] - vec[leave]) / x;`	ub: 24.3% -> 39.3% vec: 45.7% -> 87.6%
126	`else`	
127	`val = (lb[leave] - vec[leave]) / x;`	lb: 29.3% -> 40.1% vec: 42.5% -> 89.7%
128	`}`	

Figure 4.14: Code example from Soplex.

front-end has a large invocation overhead; we expect that this latency can be optimized by 15× [71].

Voyager also enjoys a dramatically lower storage overhead than the embedding-based Delta-LSTM because of its hierarchical structure. In particular, since the storage cost for neural-prefetchers is dominated by the embedding layer, the hierarchical structure reduces the model storage by 20-56×.

To further reduce the storage of Voyager, we apply standard pruning and quantization methods. We find that 80-85% of weights can be pruned with minimal accuracy loss, leading to an additional compression of 5-7×. Quantization from 32 bits to 8 bits can provide another 4× compression with minimal accuracy loss (less than 1%). Together, both optimizations enable Voyager to realize 110-200× improvements in storage compared to Delta-LSTM. Significantly, after these opti-

mizations, Voyager is 5-10× smaller than conventional temporal prefetchers, such as STMS, Domino and ISB.

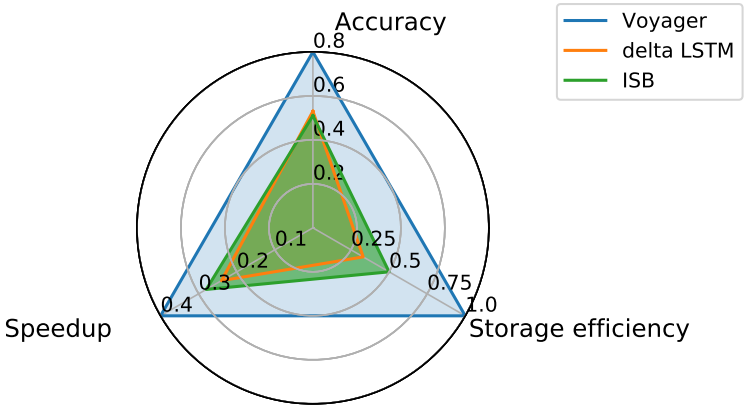


Figure 4.15: Voyager wins on accuracy, speedup, and storage efficiency. Here storage efficiency is log-scaled and defined as $\frac{1}{1+log_{10}(storage)}$

To summarize, Figure 4.15 shows that Voyager outperforms ISB and Delta-LSTM along multiple dimensions.

4.4.6 Model Specifications

For better reproducibility, we report all the hyperparameters used in Voyager in Table 4.3.

4.4.7 Paths to Practicality

We see multiple paths to building a practical prefetcher from Voyager. First, future work can use insights gained from Voyager to build a practical prefetcher without a complex neural network. For example, Glider [107] showed how to replace LSTMs with simple perceptrons for cache replacement.

Table 4.3: Hyperparameters for training Voyager.

Sequence length	16
Learning rate	0.001
Learning rate decay ratio	2
Embedding size for PC	64
Embedding size of page	128
Embedding size of offset	12800
Page and offset LSTM # layers	1
Page and offset LSTM # units	256
Dropout keep ratio	0.8
Batch size	256
Optimizer	Adam

On example of such an insight is that by using longest matching [103] on the address sequence, a table-based solution can approach 70-75% of the benefits of Voyager. The advantage of table-based solution is that tables do not involve the computational overhead of neural network that is hard to optimize. Further, we find that the both PC-localized stream and the global stream provide useful patterns, and as shown in Figure 4.16, performing longest matching on both address streams (with degree 2) provide even more benefits than Voyager (with degree 1). To optimize the storage overhead, we have an importance observation that ignoring the order of the longest matched patterns only hurts the IPC by 0.2% but reduces the storage by 33.4%.

Second, there are indications that neural networks itself can be made more computationally practical. For example, few shot learning [124] can reduce the size of training data by 20-80×, and we estimate that hierarchical softmax [84] will reduce both training and inference time by 3-5× by further reducing the number

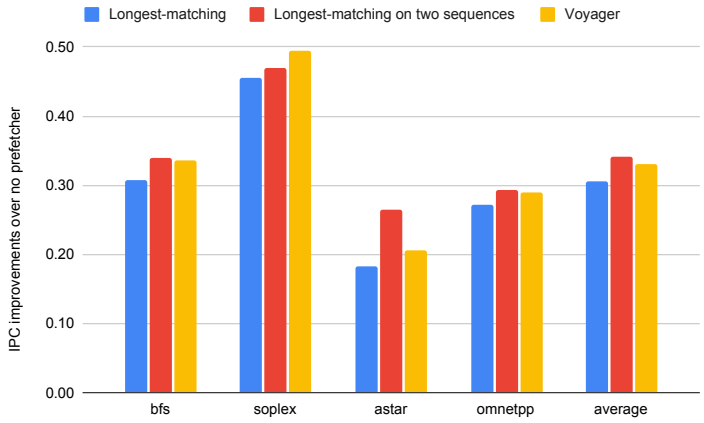


Figure 4.16: Longest matching on a single sequence and two sequences (global and PC-localized) approximate Voyager.

of classes. Related work also shows that a low-level optimized implementation of neural networks without Tensorflow's python interface can improve the prediction latency by 15× in a similar setting [71].

It is also possible to train application-specific neural prefetchers offline, particularly for workloads such as Search or Ads where rule-based prefetchers are ineffective. For example, recent work proposed an offline-trained neural network branch predictor with OS and ISA interfaces that allow the runtime system to pass parameters to the hardware [81].

4.5 Summary

In this section, we have created a probabilistic model of data prefetching in terms of features and localization. We have then presented a new neural model of data prefetching that accommodates both delta patterns and address correlation.

The key to accommodating address correlation is our hierarchical treatment of data addresses: We separate the addresses into pages and offsets, and our model makes predictions for them jointly. Our neural model shows that there is significant headroom for data prefetchers. For a set of irregular SPEC and graph benchmarks, our model achieves 79.6% accuracy/coverage and improves IPC over a baseline with no prefetching by 41.6%, compared with 57.9% and 28.2%, respectively, for an idealized ISB prefetcher. We also present results for two important commercial programs, Google's Search and Ads, which until now have seen little benefit from any data prefetcher. Voyager gets 37.8% coverage for Search (13.8% for ISB) and 57.5% for Ads (26.2% for ISB). Voyager's success comes from its ability to use data addresses as a feature and from its ability to use multiple localization schemes.

Work remains in further reducing the computational costs of our neural prefetcher. And even if literal neural models remain impractical, their insights will guide the development of practical prefetchers, in terms of features and localization schemes.

Chapter 5

Hardware-Software Co-Design of Neural Accelerator with Bayesian Optimization

Hardware/software co-design typically performed manually, but we believe that this vast design space is best navigated by an intelligent search process. To facilitate this automation, we need a formal representation of the design space.

5.1 A Formal Representation of Software and Hardware

This section formally defines the hardware and software design spaces.

5.1.1 Parameterizing the Design Space

Software design points can be parameterized by the loop ordering, loop tiling, and computational parallelism of the seven-level loop nest used to compute a convolutional layer (see Figure 5.1), as has been noted by recent work [90, 135]. These software parameters are subject to hardware constraints, such as the quantity and layout of processing elements (PEs) and the size of storage elements.

Hardware parameters are generally more specific to the low-level resource and memory configurations or the layout of PEs. These can be broken down into a two broad categories:

```
for n in [0:N)
  for k in [0:K)
    for r in [0:R)
      for s in [0:S)
        for p in [0:P)
          for q in [0:Q)
            for c in [0:C)
              outputs[n][k][q][p] += weights[k][c][s][r] *
                                     inputs[n][c][q+s][p+r]
```

Figure 5.1: Computing a 2D convolution with a seven-level nested loop.

Resource configurations represent the physical aspects of hardware, such as buffer sizes, tile sizes, and the cluster size of global buffers, as well as the layouts of the PE array and of the global buffer.

Dataflow configurations represent the usage of the PE array that are implemented in hardware, such as the blocking factors and degree of parallelism at the PE level, which also determines the communication patterns among PEs.

Figure 5.2 shows two possible design points for a 1D convolution. Both design points tile and parallelize the channel (C) dimension. To the right of each component in the architecture is a set of loops that specifies the control logic for the component, which can be broken down into temporal streaming (`for` loops) and spatial distribution (`parallel_for` loops). For example, in the architecture on the left, the global buffer distributes across the PEs 1 weight from 4 separate channels (`c2`), and the PEs perform all operations that the weight participates in. In this design, all data reuse is captured within the PE, so the global buffer need not store anything. By contrast, the architecture on the right distributes a single output

element across the PEs to compute partial sums, which are stored in the global buffer across iterations. Both these design points consist of the same architectural components, but the dataflows vary, imposing different constraints on the software.

To show the details of the parameterization of a more practical 2D convolution, Figure 5.3 shows a design point for the CONV4 layer of ResNet. The architecture components are again the same as in the 1D example, but since the memory footprint is significantly larger, the PE can no longer capture all data reuse, so the Global Buffer must store large portions of the inputs and outputs.

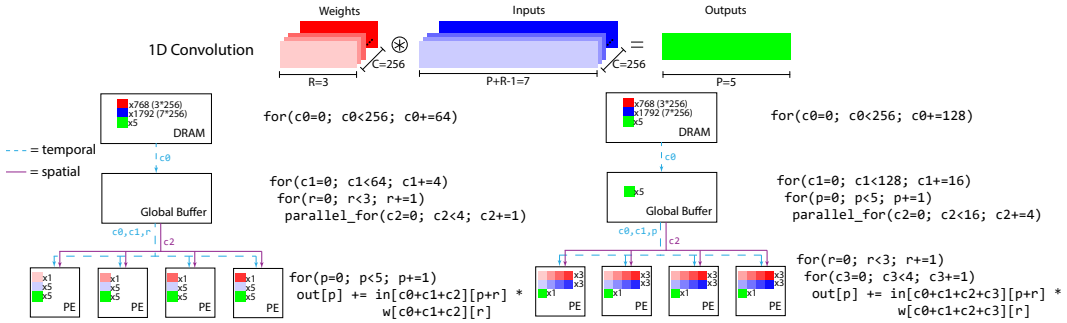


Figure 5.2: Two architectures computing a 1D convolution.

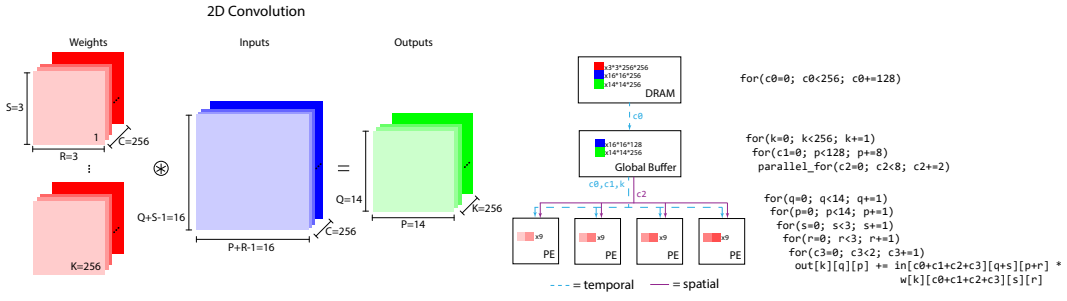


Figure 5.3: An architecture computing the CONV4 layer of ResNet.

5.1.2 Constraints in the Design Space

There are several reasons why the vast space of hardware and software parameters is filled with impractical or invalid design points. First, hardware designs are fundamentally constrained by area (the total amount of compute and storage resources) and factors such as available memory bandwidth. Second, the design cost and latency of additional area grow super-linearly [105], which leads to many impractical design points.

Software constraints are generally governed by feasibility instead of practicality and predominantly depend on the hardware configuration and the specific neural network workload. For a specific hardware accelerator, there is a limited number of available resources, so the software optimization problem can be viewed as a search for the most efficient use of hardware PEs and buffers. For example, the loop blocking optimization factors a neural network across multiple hardware storage buffers—and the feasible factorizations are constrained by the size of the hardware buffers.

5.2 Bayesian Optimization

5.2.1 Overview

Bayesian optimization [62, 12, 104] is an effective approach for the optimization of expensive, possibly noisy black-box functions. BO has been used to optimize hyperparameters [113], configure algorithms [45], optimize A/B experiments [77], and more. For our problem, we have a parameterized representation and access to a simulator. Since one of our main concerns is sample efficiency, Bayesian

optimization is particularly suitable.

The actual cost of evaluation depends on the experimental infrastructure, but in general, it is much more expensive to evaluate a hardware design choice than to evaluate software optimizations, because hardware design can take hours (to produce a hardware simulator or an FPGA) to days or even months (to produce an ASIC).

Bayesian optimization has two major components: (1) a surrogate model provides a Bayesian posterior probability distribution that predicts potential values of the objective function. (2) an acquisition function uses the model to identify the next point to evaluate.

5.2.2 Gaussian processes

A common surrogate model is a Gaussian process (GP) [98] due to its simplicity and flexibility. A GP is prior distribution over the space of functions comprised of a mean function $m(\mathbf{x})$ and a covariance, or kernel function $k(\mathbf{x}, \mathbf{x}')$. Suppose we are given a dataset of N input/output pairs over a bounded domain Ω with D input dimensions and scalar outputs. For brevity, we write this as $(X, \mathbf{y})$, where $X \in \Omega^{N \times D}$ and $\mathbf{y} \in \mathbb{R}^N$. The posterior predictive distribution over function values f for a new input $\mathbf{x}$ is given by $P(f \mid \mathbf{x}, X, \mathbf{y}) = \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))$, where

$$\mu(\mathbf{x}) = K_{\mathbf{x}X} K_{XX}^{-1} (\mathbf{y} - \mathbf{m}_X) + m(\mathbf{x}),$$
$$\sigma^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - K_{\mathbf{x}X} K_{XX}^{-1} K_{\mathbf{x}X}^{\top}.$$

Where K_{XX} is a matrix formed by evaluating the kernel on X, $K_{\mathbf{x}X}$ is the vector of kernel evaluations between $\mathbf{x}$ and X, and $\mathbf{m}_X$ is the vector of mean function evaluations on the input dataset.

A common choice for the kernel is squared exponential. Given two input vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, this is defined as $k(\mathbf{x}_i, \mathbf{x}_j) = \alpha^2 \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\boldsymbol{\ell}^2}\right)$. α and $\boldsymbol{\ell}$ are kernel hyperparameters.

Another kernel that we find particularly useful is a linear kernel on top of explicit features. Given a feature mapping $\phi(\mathbf{x}) : \mathbb{R}^D \to \mathbb{R}^K$, the linear kernel can be written as $k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$. When we have strong prior information about the relevant feature interactions that govern the black-box function, this kernel allows us to encode these interactions directly and produces a more sample-efficient posterior estimate.

In cases where the observations from the black-box function are noisy, we can add a noise kernel $K_{\text{noise}} = \tau^2 \mathrm{I}$ to K_{XX}, where τ^2 is a hyperparameter. This implies a Gaussian observation likelihood.

Following common practice, we use the constant mean $m(\mathbf{x}) = c \quad \forall\, \mathbf{x}$. All kernel and mean hyperparameters are learned by maximizing the marginal likelihood of the GP on the current dataset.

5.2.3 Acquisition functions

A critical component in the BO framework is the choice of acquisition function $a(\cdot)$ that assigns each design point a value that represents the utility of testing

this point. Two commonly used acquisition functions are expected improvement (EI) and lower confidence bound (LCB).

EI computes the amount we expect to improve upon the current best observed objective value $y_* \equiv \max\{y_i\}_{i=1}^N$ by evaluating a design point $\mathbf{x}$. Formally, it can be written as

$$a_{\mathrm{EI}}(\mathbf{x}) = \int_{-\infty}^{\infty} \max(y_* - f, 0) P(f \mid \mathbf{x}, X, \mathbf{y}) \mathrm{d}f.$$

where f is the latent function from the surrogate model, and y_* is the best value observed.

LCB [116] provides an explicit tradeoff between the predictive mean and variance and is defined as

$$a_{\mathrm{LCB}}(\mathbf{x}) = \mu(\mathbf{x}) + \lambda\sigma(\mathbf{x}).$$

Where λ represents a tradeoff parameter. A small λ promotes greater exploitation, and a large λ promotes greater exploration. We found $\lambda = 1$ to work well in our experiments. Beyond these, there are many other possible acquisition functions that could be used in future exploration [122, 36, 39, 26].

5.2.4 Constraints

In our problem, the vast majority of the design space will produce invalid solutions. When the constraints are a known function of the input features, we can directly account for them as input constraints. Otherwise, we must run the simulation and treat invalid points using an output constraint. Here, we will describe these constraint types, and how they are incorporated into BO.

Input constraints are explicit constraints that are used when optimizing the acquisition function. They directly prevent the search from suggesting points that will violate the constraints. As some constraints are non-linear, this optimization is itself very challenging, as it is a global optimization problem with non-convex constraints. In the unconstrained case, maximizing the acquisition function often takes a hybrid approach: generating a random initial set of points and refining them by gradient ascent. Maintaining feasibility with non-convex constraints is far more challenging, however.

We therefore optimize the acquisition function in a simple way by performing rejection sampling on the design space: we randomly sample parameters until we obtain 150 feasible points, and then choose the one the maximizes the acquisition function. On average the sampling takes 22K random samples to get a pool of 150 feasible points. We have found that practically this is a simple yet effective strategy for our problems, we leave more involved optimization schemes for future work.

Output constraints are used when we do not know the form of the constraint a-priori and must run the simulator to test feasibility. This is also referred to as an "unknown" constraint, and BO has been adapted to incorporate a constraint model in addition to the regression model [29]. These simultaneously learn about the constraint boundaries while modeling the objective.

Let $\mathcal{C}(\mathbf{x})$ denote the event that $\mathbf{x}$ satisfies constraint $\mathcal{C}$. Constrained BO uses a Bayesian classifier to model $P(\mathcal{C}(\mathbf{x}))$. It is relatively straightforward to adapt a GP regressor to classification [98]. In our case, observations come in the form of

binary observations, 1 if the observation is feasible and 0 otherwise. A GP models this by assuming a real valued function $f(\mathbf{x})$ is drawn from a GP prior, which is then transformed through a link function (e.g., sigmoid or probit) to form a Bernoulli probability. Inference is no longer closed-form, but can be efficiently approximated through variational inference, expectation propagation, or Monte-carlo sampling [98].

Under a Bayesian classifier, the acquisition function $a(\mathbf{x})$ is modified to account for the probability that the constraint is satisfied, with 0 utility if it is not satisfied.

$$\bar{a}(\mathbf{x}) = \mathbb{E}[a(\mathbf{x})\mathrm{I}[\mathcal{C}(\mathbf{x})]] = P(\mathcal{C}(\mathbf{x}))a(\mathbf{x}).$$

Where $\mathrm{I}[\mathcal{C}(\mathbf{x})]$ is the indicator function that evaluates to 1 if the constraint is satisfied and 0 otherwise. We therefore maintain two models: one regression model to capture the objective and one classifier to model the constraint in order to avoid evaluations in infeasible regions.

5.3 Bayesian Optimization for Hardware/Software Co-design

5.3.1 Overview of Nested Hardware/Software Optimization

Provided the constraints discussed in Section 5.1 and the BO formulation from Section 5.2, we propose a nested approach for co-optimizing hardware/software parameters. The overall approach is outlined in Figure 5.4. The goal is to find the optimal hardware parameters for a neural model and the optimal set of software parameters for each layer in the neural model. Since software constraints depend on

a feasible hardware design, we first propose the hardware parameters, then for that hardware co-optimize the software mapping.

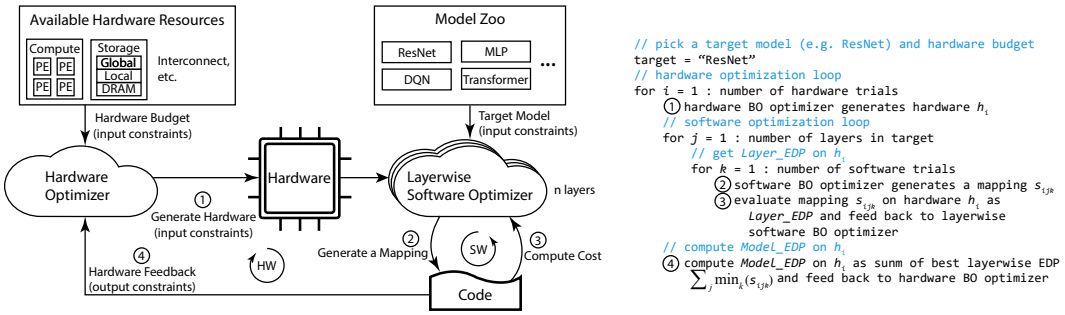


Figure 5.4: Overview of BO-based nested search for hardware/software co-design.

Specifically, let $\mathbf{x}_h$ and $\mathbf{x}_s$ denote the set of hardware and software parameters in the parameter space to be optimized. In the nested search process, we first use the hardware optimizer to generate a design of hardware. In particular, we perform the hardware search in the space of possible hardware $\mathcal{S}_h$ to optimize all hardware parameters, where the objective is to minimize $f(\mathbf{x}_h \mid \text{NN})$ which we define as the energy-delay product (EDP) of running the neural network (NN) model on the given hardware, assuming the optimal software mapping for each individual layer. This step produces a hardware specification and can be formalized as $\text{argmin}_{h \in \mathcal{S}_h} f(\mathbf{x}_h \mid \text{NN})$.

For the chosen hardware design, our framework performs the software search for each individual neural layer in its constrained software mapping space $\mathcal{S}_s | h, \text{NN}_j$ to optimize the mapping parameters, where NN_j denotes the jth layer in the neural network model, and the objective becomes $f(\mathbf{x}_s \mid \mathbf{x}_h, \text{NN}_j)$, which is defined as the EDP of running the layer j on the fixed hardware. This step produces

a design point that represents the best set of software mappings for each layers on the given hardware structure, and can be formalized as $\text{argmin}_{s \in \mathcal{S}_s|h} f(\mathbf{x}_s \mid \mathbf{x}_h)$. The layerwise EDPs are then summed up as the EDP of the neural model, which is fed back to the hardware optimizer to generate the next hardware setting.

The iterative search between hardware and software will repeat for a user-defined number of trials. In this work, we set 50 for hardware search and 250 for software search. The combination of hardware and software that achieves the best EDP during the optimization process becomes the final model-specific hardware structure and layer-specific software mappings. A random sample is used in the first iteration of both the hardware and software search. In our Bayesian optimization (BO) framework, we use separate BO models to search in the hardware and software space. In Figure 5.5 we report the hyperparamters for BO. We now describe their design considerations, particularly the choice of kernel and feature transformation.

number of independent trials	5 (HW), 10 (SW)
number of random data points	50 (HW), 150 (SW)
number of warmup data points	5 (HW), 30 (SW)
number of samples for EI	1000
lambda for LCB	1.0

Figure 5.5: Hyperparamters for BO.

5.3.2 BO for Optimizing Hardware Architectures

Kernel design. The main design choice for BO is the GP kernel to use. For the hardware search, we choose a linear kernel on top of feature transformations that represent the relationship between the different parameters. This feature transformation allows us to explicitly encode domain knowledge. For example, by comparing the layout parameters of the 2D PE array and global buffer we can obtain the ratio between these adjacent storage layers, which correlates to the maximal degree of parallel buffer accesses in each dimension. The details of the features are given in Figure 5.8. We also add a noise kernel to deal with noise in the hardware evaluation. This is because the software optimizer is not guaranteed to find the best software mapping for each layer. There is some randomness in the software search, and therefore independent runs of software optimization for a fixed hardware design may yield different results.

Constraints. There are both known and unknown constraints in the hardware search. The known constraints, such as the compute and storage budget, are treated as input constraints that reject invalid samples. The unknown constraints have to do with feasibility (if there exist valid software mappings of neural layers onto the hardware, and if the valid mappings can be sampled during the software optimization). Following Section 5.2, these constraints are treated as output constraints and are modeled by a GP with a squared exponential kernel.

Type	Index	Hardware Parameters	Valid Range	Meaning
PE	H1	PE mesh-X	Factors of # PEs	Decide the arrangement of the 2-D PE array.
	H2	PE mesh-Y	Factors of # PEs	
Local buffer	H3	Input entries in Local buffer	0 to # local buffer entries	Decide the partition of local buffer. The partition leads to sub-buffers with inflexible sizes. This is useful as the latency to access each smaller sub-buffer decreases.
	H4	weights entries in Local buffer	0 to # local buffer entries	
	H5	outputs entries in Local buffer	0 to # local buffer entries	
Global buffer	H6	Global buffer instances	Factors of #PEs	Determine the arrangement of global buffer, and its connection between global buffer and per PE's local buffer (Local buffer of PEs along the X-axis shares the instances of global buffer along the X-axis).
	H7	Global buffer mesh-X	Factors of PE-mesh-X	
	H8	Global buffer mesh-Y	Factors of PE-mesh-Y	
	H9	Global buffer block size	Factors of 16	Determines the width of a global buffer entry
	H10	Global buffer cluster size	Factors of 16	Determines of the number of wider structures where multiple entries are ganged into
Dataflow	H11	Dataflow option of filter width	1, 2	Options that determine the size of filter width in PE's local buffer
	H12	Dataflow option of filter height	1, 2	Options that determine the size of filter height in PE's local buffer

Figure 5.6: Hardware parameters.

5.3.3 BO for Optimizing Software Mappings

Kernel design. Similar to hardware optimization, we use a linear kernel and transform the parameters to features that encode relational information. As the hardware is fixed in the search of software mappings, we are able to compute features such as buffer usage, which potentially help make the predictions more accurate. The evaluation of a mapping on a given hardware is deterministic in our infrastructure, thus there is no need for a noise kernel in the GPs.

Type	Hardware Constraints
PE	PE mesh-X (H1) * PE mesh-Y (H2) = # PEs
Local buffer	The sum of local sub-buffers (H3, H4, H5) does not exceed buffer size
Global buffer	Global buffer mesh-X (H7) * global buffer mesh-Y (H8) = # Global buffer instances (9)
Local buffer & global buffer (unknown)	A valid software mapping exists depending mainly on local buffer partition (H3, H4, H5) and global buffer arrangement (H6, H7, H8)

Figure 5.7: Hardware constraints.

Model	Feature name	Description
Hardware	mesh_x_ratio	The ratio of PE array and global buffer along x-axis
	mesh_y_ratio	The ratio of PE array and global buffer along y-axis
Software	input_buffer_usage	input data size / input (local) buffer size
	weight_buffer_usage	weight data size / input (local) buffer size
	output_buffer_usage	output data size / input (local) buffer size
	global_buffer_usage	all data size / global buffer size
	parallelism_ratio_x	used parallelism / available parallelism in the x-axis of global buffer
	parallelism_ratio_y	used parallelism / available parallelism in the y-axis of global buffer

Figure 5.8: Extra features used by the hardware and software BO optimizers.

Constraints. As both the hardware and neural model are known during software optimization, all constraints are known and are treated as input constraints that automatically reject invalid samples.

5.4 Evaluation

5.4.1 Methodology

Infrastructure. We conduct our evaluation on Timeloop [90], which is an open-source infrastructure for evaluating the hardware design and software optimization of DNN accelerators. Timeloop represents the key architecture attributes of DNN accelerators that realize a broad space of hardware structure and topology, which generate an accurate projection of performance and energy efficiency for

Type	Index	Software Parameters	Valid Range	Meaning
Loop blocking and degree of parallelism	S1	Blocking factors of R	Factors of R	Determines the size (parallelism) of each type of data (inputs, weights and outputs) in each storage layer (except those that are in the hardware dataflow).
	S2	Blocking factors of S	Factors of S	
	S3	Blocking factors of P	Factors of P	
	S4	Blocking factors of Q	Factors of Q	
	S5	Blocking factors of C	Factors of C	
	S6	Blocking factors of K	Factors of K	
Loop reorder	S7	Loop order in local buffer	Permutations of non-1 factors	Affects the reuse of each type of data (inputs, weights and outputs) in each storage layer.
	S8	Loop order in global buffer	Permutations of non-1 factors	
	S9	Loop order in DRAM	Permutations of non-1 factors	

Figure 5.9: Software parameters.

DNN workloads. In the evaluation, Timeloop takes two inputs: 1) the hardware configuration, which consists of the hardware-related parameters, and 2) the software mapping, which consists of the software parameters that describe the mapping. As most accelerators are designed for neural network inference, we limit the use case to inference in this work and leave training for future work.

Workloads. One of the goals of the work is to demonstrate that the optimizer can automatically co-design optimal hardware for a variety of neural network models without human effort. Therefore, we use our BO framework to optimize critical layers from CNNs (ResNet [34] and DQN [85]), as well as an MLP and Transformer [123]. In Figure 5.11 and Figure 5.12 we report the specifications of neural models benchmarked in this work.

Type	Software Constraints
Loop blocking and degree of parallelism	Product of all blocking factors of R (S1) equals R of the target neural layer
	Product of all blocking factors of S (S2) equals S of the target neural layer
	Product of all blocking factors of P (S3) equals P of the target neural layer
	Product of all blocking factors of Q (S4) equals Q of the target neural layer
	Product of all blocking factors of C (S5) equals C of the target neural layer
	Product of all blocking factors of K (S6) equals K of the target neural layer
Buffer capacity (local)	Inputs/weights/outputs sizes (S1-S6) cannot exceed corresponding local sub-buffer capacity
Buffer capacity (global)	Size of all types of data (S1-S6) does not exceed global buffer capacity
Parallelism	Product of blocking factors in global buffer X-axis (S1-S6) cannot exceed # PEs in X-axis
	Product of blocking factors in global buffer (S1-S6) cannot exceed total # PEs

Figure 5.10: Software constraints.

Experimental Setup. We use Eyeriss [17], a state-of-the-art DNN accelerator, as our baseline. All workloads are evaluated on the Eyeriss implementation with 168 PEs [17] except for the Transformer model, which runs on the larger version of Eyeriss with 256 PEs [90]. In the software mapping search, we use Eyeriss's hardware specifications and search for the best software mapping for each neural layer. In the hardware search, we perform the search under the same compute and storage resource constraints as Eyeriss for each neural model. [1]

Metrics. Hardware accelerators are designed to achieve both speed and energy efficiency. In this work, we adopt the widely used energy-delay product (EDP) as the objective. As the actual EDP values vary across an order of magnitude, we normalize by dividing by the best (minimal) EDP value, and take the reciprocal for optimization curves. For the hardware/software co-design, we report the EDP

[1] This work focuses on model-specific hardware, but hardware specialization provides larger benefits at a finer granularity, i.e. if different layers can execute on customized hardware. We leave this for future work.

Model	Layers	Specifications
ResNet	ResNet-K1	Filter size: 3×3 Output size: 56×56 # input channel: 64 # output channel: 64 Stride: 2
	ResNet-K2	Filter size: 3×3 Output size: 28×28 # input channel: 128 # output channel: 128 Stride: 1
	ResNet-K3	Filter size: 3×3 Output size: 14×14 # input channel: 256 # output channel: 256 Stride: 1
	ResNet-K4	Filter size: 3×3 Output size: 7×7 # input channel: 512 # output channel: 512 Stride: 1
DQN	DQN-K1	Filter size: 8×8 Output size: 20×20 # input channel: 4 # output channel: 16 Stride: 4
	DQN-K2	Filter size: 4×4 Output size: 9×9 # input channel: 16 # output channel: 32 Stride: 2

Figure 5.11: Specifications of ResNet (ResNet-18) [34] and DQN [85]

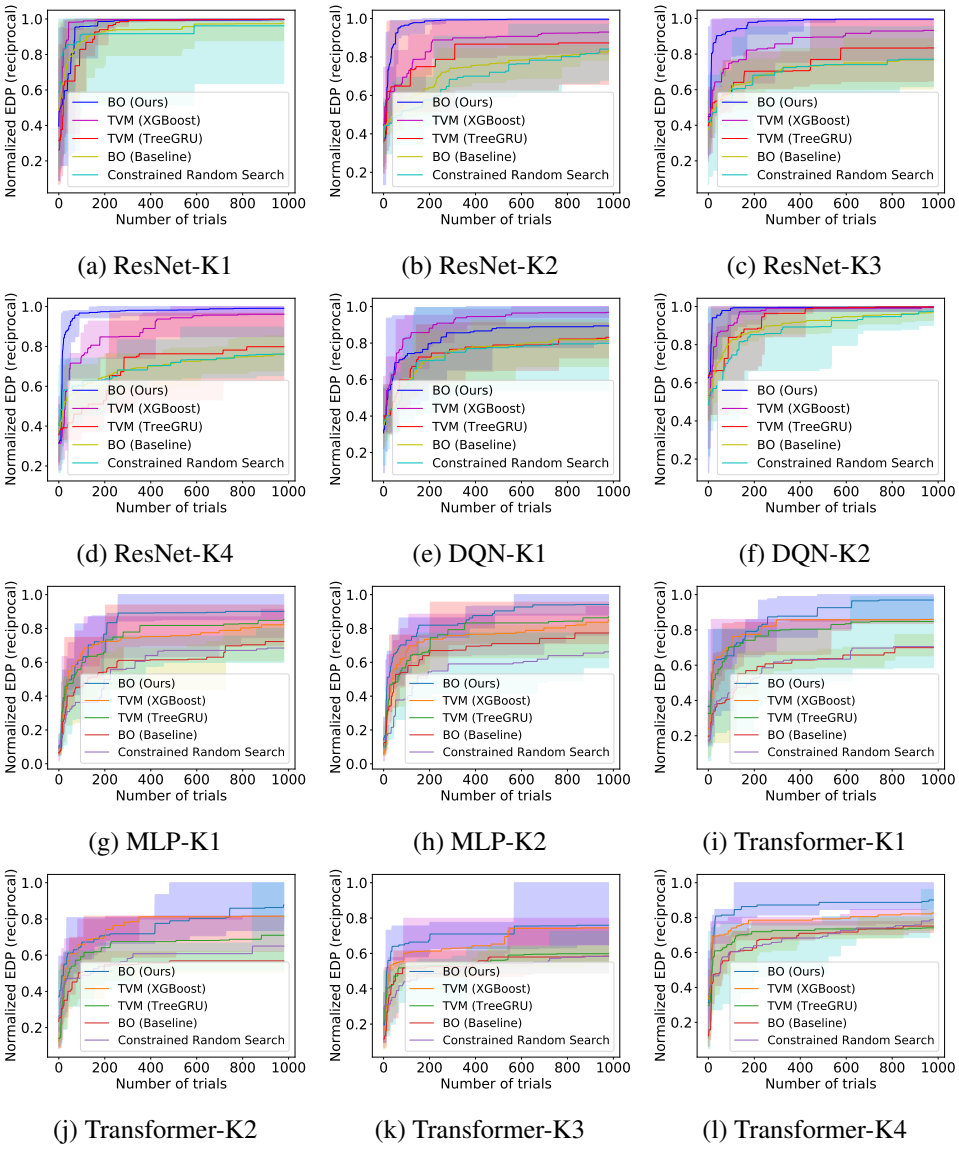
improvements of each neural model, which is averaged across all layers (see Figure 5.11 and 5.12). For software mapping optimizations, we report the layer-wise EDP

Model	Layers	Specifications
MLP	MLP-K1	d_{in}: 512 d_{out}: 512
	MLP-K2	d_{in}: 64 d_{out}: 1024
Transformer	Transformer-K1	d_{model} = 512 d_v = 32 d_k = 32 h = 16
	Transformer-K2	d_{model} = 512 d_v = 64 d_k = 64 h = 8
	Transformer-K3	d_{model} = 512 d_v = 128 d_k = 128 h = 4
	Transformer-K4	d_{model} = 512 d_v = 512 d_k = 512 h = 1

Figure 5.12: Specifications of MLP and Transformer [123]

improvements.

Baselines. In hardware search, we compare against constrained random search that repeatedly takes the first random sample in the design space that satisfies the constraints. In software search, we use both constrained random search, and out-of-the-box BO that optimizes in a continuous parameter space and rounds to the nearest valid parameters.



(a) ResNet-K1 (b) ResNet-K2 (c) ResNet-K3

(d) ResNet-K4 (e) DQN-K1 (f) DQN-K2

(g) MLP-K1 (h) MLP-K2 (i) Transformer-K1

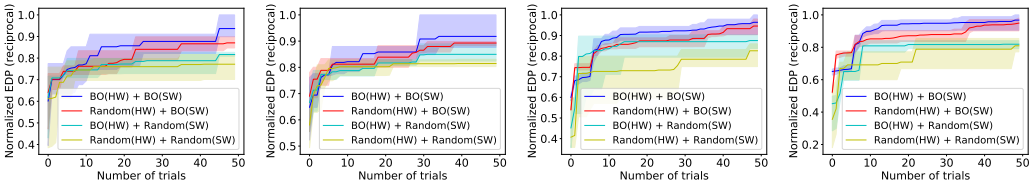
(j) Transformer-K2 (k) Transformer-K3 (l) Transformer-K4

Figure 5.13: Software mapping optimization on ResNet, DQN, MLP, and Transformer. The Y-axis shows the reciprocal of energy-delay product (EDP) (normalized against the best EDP value). Higher is better.

5.4.2 Software Mapping Optimization

We show the results of software mapping optimization first, as the capability of finding a good mapping is the base of evaluating a hardware design. Figure 5.13

shows the improvements of BO over our constrained random search formulation. Our BO formulation outperforms random search as well as a standard BO formulation that treats parameters as continuous and uses a relax-and-round approach. This result is particularly interesting when compared to using BO for hyperparameter tuning, where out-of-the-box BO works much better in a less-constrained parameter space.



(a) ResNet (b) DQN (c) MLP (d) Transformer

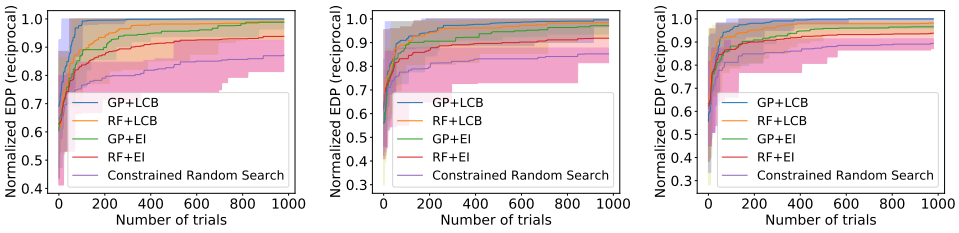
Figure 5.14: Hardware/software co-optimization. The x-axis shows the number of trials for hardware search, and 250 attempts are made to find the optimal software mapping for each layer in the model on the hardware specification. Best viewed in color.

5.4.3 Hardware Configuration Optimization

The evaluation of hardware search also involves the search of the software mapping. Figure 5.14 shows the optimization curves for hardware/software co-design. The comparison of hardware search algorithms shows that BO provides consistently better performance than the constrained random search, and the comparison of software search algorithms shows the importance of mapping optimization in the co-design process. We find that the designs searched by BO achieve significantly better EDP on all neural models compared to the state-of-the-art manually designed accelerator (18.3%, 40.2%, 21.8% and 16.0% for ResNet, DQN, MLP and Transformer respectively).

5.4.4 Ablations

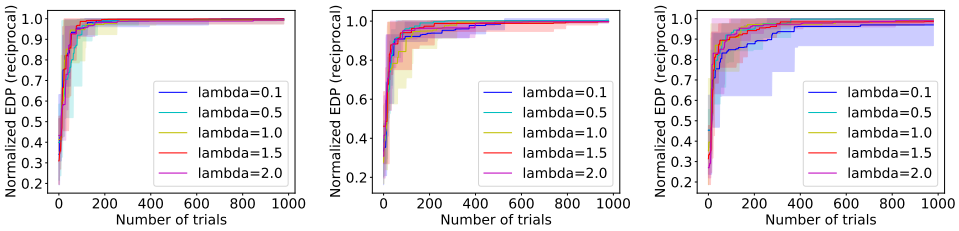
In Figure 5.15 we compare different surrogate models and acquisition functions for Bayesian optimization of the software mapping. We found Gaussian processes with LCB to consistently outperform other alternatives.



(a) ResNet-K2 (b) ResNet-K3 (c) ResNet-K4

Figure 5.15: GP with different surrogate models and acquisition functions.

In Figure 5.16 we investigate the robustness of LCB for software optimization using different values of λ. We found that $\lambda = 0.1$ tends to be too greedy, but that above $\lambda = 0.5$, LCB tends to be fairly robust.



(a) ResNet-K2 (b) ResNet-K3 (c) ResNet-K4

Figure 5.16: LCB acquisition function with different lambda values.

5.5 Summary

In this section, we have cast hardware/software co-design as a Bayesian optimization problem. We have shown that standard mechanisms have difficulty navigating the complex, highly constrained design space, so we have presented a novel constrained formulation that allows the optimizer to efficiently identify desirable points in this design space. The use of machine learning to automate hardware/software co-design opens many opportunities for future work. For example, transfer learning could dramatically reduce design time across designs and models. The techniques described here are not limited to DNN architectures, which is significant because as we enter the golden age of computer architecture [35], it is essential that we develop automatic mechanisms for architectural exploration that quickly produce custom hardware accelerators.

Chapter 6

Conclusions

In this thesis, we have presented machine learning solutions that significantly advance the state-of-the-art for three architectural problems, namely, cache replacement, data prefetching and the design automation of neural network accelerators. For cache replacement, we have introduced the Glider replacement policy that uses a long history of past memory accesses with a simple SVM model. For data prefetching, we have introduced Voyager, which is the first learning-based prefetcher that combines address and delta prediction, and provides insights to practical solutions. Finally, we have introduced a Bayesian optimization framework to automatically explore the design space of neural network accelerators. The key feature of our framework is its ability to navigate through the highly constrained space.

Prior to this thesis, the field of computer architecture has seen some success in the use of off-the-shelf machine learning models, such as perceptron. But while the linear perceptron model was sufficiently simple to be practically deployed in hardware—after some hardware optimizations—the constraints of hardware prediction problems prevent the direct use of more powerful learning algorithms, which are potentially more beneficial. A unifying theme of this thesis is that our solutions are not limited to simple machine learning algorithms in their off-the-shelf uses.

Instead, our solutions handle the constraints of hardware prediction problems by innovating in the machine learning space, which provides substantial improvements by enabling more powerful learning algorithms, such as deep learning.

Looking to the future, this thesis opens up broad research questions. First, can our offline-to-online approach be applied to other hardware predictors beyond cache replacement and data prefetchers? Most of the other important predictors, such as TLB prefetching/management, DRAM scheduling, and memory disambiguation, are still dominated by heuristic-based solutions. We believe that our offline-to-online approach offers exciting opportunities for future research of both computer architecture and machine learning. Second, our Bayesian optimization framework sheds light on significantly improving the efficiency of design exploration with machine learning. Future work can apply our framework to other problems, such as routing and scheduling, or further improve on our framework in a more realistic hardware setting (e.g. FPGA).

In summary, this thesis has shown successful applications of machine learning to three hardware prediction problems, where the unique constraints can be effectively handled through machine learning innovations. Much of success in the future of this area will require innovations in the machine learning space, which provides challenging interdisciplinary research questions, as well as the potential to make substantial benefits. We believe that our design philosophy is extensible to many other hardware predictors and design exploration problems in computer architecture.

Bibliography

[1] "The 2nd cache replacement championship," 2017. [Online]. Available: http://crc2.ece.tamu.edu/

[2] "NVIDIA Tesla V100 GPU Architecture, The World's Most Advanced Data Center GPU," *NVIDIA Corporation*, 2017.

[3] J. Abella, A. González, X. Vera, and M. F. O'Boyle, "Iatac: a smart predictor to turn-off l2 cache lines," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 1, pp. 55–77, 2005.

[4] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, May 1995.

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

[6] 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)*. IEEE, 2019, pp. 399–411.

[7] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[8] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, pp. 78–101, 1966.

[9] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 531–544.

[10] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 1–13.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.

[12] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[13] D. Burger, T. R. Puzak, W.-F. Lin, and S. K. Reinhardt, "Filtering superfluous prefetches using density vectors," in *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, 2001, pp. 124–133.

[14] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04, 2004, pp. 276–288.

[15] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.

[16] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[17] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[18] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[19] T. M. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001, pp. 191–202.

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

[21] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 92–104.

[22] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *45th International Symposium on Microarchitecture*, ser. MICRO, 2012, pp. 389–400.

[23] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in *26th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, 2017, pp. 180–193.

[24] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Cvpr 2011 Workshops*. IEEE, 2011, pp. 109–116.

[25] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 133–143.

[26] P. I. Frazier, "Knowledge-gradient methods for statistical learning," Ph.D. dissertation, Citeseer, 2009.

[27] H. Gao and C. Wilkerson, "A dueling segmented LRU replacement algorithm with adaptive bypassing," in *JWAC 2010-1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[28] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.

[29] M. A. Gelbart, J. Snoek, and R. P. Adams, "Bayesian optimization with unknown constraints," *Uncertainty in Artificial Intelligence*, 2014.

[30] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *27th International Symposium on Computer Architecture (ISCA)*, 2000, pp. 107–116.

[31] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, 2005.

[32] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[33] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *35th International Conference on Machine Learning*, ser. ICML, 2018, pp. 1924–1933.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[35] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[36] P. Hennig and C. J. Schuler, "Entropy search for information-efficient global optimization," *Journal of Machine Learning Research*, vol. 13, no. Jun, pp. 1809–1837, 2012.

[37] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, pp. 1–17, 2006.

[38] D. Hernandez and T. B. Brown, "Measuring the algorithmic efficiency of neural networks," 2020.

[39] J. M. Hernández-Lobato, M. W. Hoffman, and Z. Ghahramani, "Predictive entropy search for efficient global optimization of black-box functions," in *Advances in neural information processing systems*, 2014, pp. 918–926.

[40] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury *et al.*, "Deep neural networks for

acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.

[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[42] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behavior," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 2002, pp. 209–220.

[43] Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: tag correlating prefetchers," in *HPCA*, 2003, pp. 317–326.

[44] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Proceedings of the 39th International Symposium on Microarchitecture*, 2006, pp. 397–408.

[45] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*. Springer, 2011, pp. 507–523.

[46] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[47] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ACM SIGARCH Com-*

puter Architecture News, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 39–50.

[48] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," in *Journal of Instruction-Level Parallelism*, vol. 13, 2011, pp. 1–24.

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

[50] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2013.

[51] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 247–259.

[52] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *43nd Annual International Symposium on Computer Architecture*, ser. ISCA. IEEE, 2016, pp. 78–89.

[53] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 110–123.

[54] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A Pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual*

Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA, 2008, pp. 28–36.

[55] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *37th International Symposium on Computer Architecture (ISCA)*. ACM, 2010, pp. 60–71.

[56] D. A. Jiménez, "Multiperspective perceptron predictor."

[57] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the 33th Annual International Symposium on Microarchitecture*, December 2000, pp. 67–76.

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

[59] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 436–448.

[60] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2017, pp. 436–448.

[61] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu, "Run-time spatial locality detection and optimization," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997, pp. 57–64.

[62] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global optimization*, vol. 13, no. 4, pp. 455–492, 1998.

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

[64] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *International Symposium on Computer Architecture (ISCA)*, 1990, pp. 364–373.

[65] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[66] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, no. 3, pp. 38–46, 1994.

[67] S. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *43rd International Symposium on Microarchitecture (MICRO)*, 2010, pp. 175–186.

[68] M. Kharbutli and Y. Solihin, "Counter-based cache replacement algorithms," in *International Conference on Computer Design (ICCD)*, 2005, pp. 61–68.

[69] C. S. Kim, "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computers*, pp. 1352–1361, 2001.

[70] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 380–392, 2016.

[71] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," 2018. [Online]. Available: https://arxiv.org/abs/1712.01208

[72] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[73] S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," *SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 357–368, April 1998.

[74] H. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," in *Sparse Matrix Proceedings 1978*, vol. 1. Society for industrial and applied mathematics, 1979, pp. 256–282.

[75] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[76] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1. ACM, 1999, pp. 134–143.

[77] B. Letham, B. Karrer, G. Ottoni, E. Bakshy *et al.*, "Constrained bayesian optimization with noisy experiments," *Bayesian Analysis*, vol. 14, no. 2, pp. 495–519, 2019.

[78] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *41st International Symposium on Microarchitecture (MICRO)*, 2008, pp. 222–233.

[79] J. Lu, J. Yang, D. Batra, and D. Parikh, "Hierarchical question-image co-attention for visual question answering," in *Advances In Neural Information Processing Systems*, 2016, pp. 289–297.

[80] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[81] S. Z. S. P. S. Lym and Y. N. Patt, "Branchnet: Using offline deep learning to predict hard-to-predict branches," 2019.

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

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

[84] A. Mnih and G. E. Hinton, "A scalable hierarchical distributed language model," in *Advances in neural information processing systems*, 2009, pp. 1081–1088.

[85] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[86] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–11, 2016.

[87] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *IEEE PACT*, 2004, pp. 135–145.

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

[89] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, April 1994, pp. 24–33.

[90] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 304–315.

[91] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.

[92] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 285–297.

[93] D. D. Penney and L. Chen, "A survey of machine learning applied to computer architecture design," *arXiv preprint arXiv:1909.12373*, 2019.

[94] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe

run-time evaluation of aggressive prefetchers," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014.

[95] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *34th International Symposium on Computer Architecture (ISCA)*. ACM, 2007, pp. 381–391.

[96] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *33rd International Symposium on Computer Architecture*, ser. ISCA, 2006, pp. 167–178.

[97] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, 2019.

[98] C. E. Rasmussen and C. K. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[99] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *the ACM Conference on Measurement and Modeling Computer Systems (SIGMETRICS)*, 1990, pp. 134–142.

[100] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962.

[101] S. Sair, T. Sherwood, and B. Calder, "A decoupled predictor-directed stream prefetching architecture," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 260–276, March 2003.

[102] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *the 21st Int'l Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 355–366.

[103] A. Seznec, "A 256 kbits l-tage branch predictor," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, pp. 1–6, 2007.

[104] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

[105] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, and P. Raina, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[106] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chisthi, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 141–152.

[107] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 413–425.

[108] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A neural hierarchical sequence model for irregular data prefetching."

[109] Z. Shi, C. Sakhuja, M. Hashemi, K. Swersky, and C. Lin, "Learned hardware/software co-design of neural accelerators," *arXiv preprint arXiv:2010.02075*, 2020.

[110] Z. Shi, K. Swersky, D. Tarlow, P. Ranganathan, and M. Hashemi, "Learning execution through neural code fusion," *arXiv preprint arXiv:1906.07181*, 2019.

[111] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: simple and effective adaptive page replacement," in *ACM SIGMETRICS Performance Evaluation Review*, 1999, pp. 122–133.

[112] A. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Transactions on Computers*, vol. 11, no. 12, pp. 7–12, December 1978.

[113] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

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

[115] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ISCA '06: Proceedings of the 33th Annual International Symposium on Computer Architecture*, 2006, pp. 252–263.

[116] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," *arXiv preprint arXiv:0912.3995*, 2009.

[117] 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*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 461–470. [Online]. Available: https://doi.org/10.1145/3357526.3357549

[118] R. Subramanian, Y. Smaragdakis, and G. H. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *39th International Symposium on Microarchitecture*, ser. MICRO. IEEE Computer Society, 2006, pp. 385–396.

[119] M. Takagi and K. Hiraki, "Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches," in *Proceedings*

of the 18th annual international conference on Supercomputing. ACM, 2004, pp. 20–30.

[120] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," *arXiv preprint arXiv:1906.09889*, 2019.

[121] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO. IEEE, 2016, pp. 1–12.

[122] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933.

[123] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[124] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–34, 2020.

[125] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal streams in commercial server applications," in *IISWC*, 2008, pp. 99–108.

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

[127] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *HPCA*, 2009, pp. 79–90.

[128] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998, pp. 38–38.

[129] W. A. Wong and J.-L. Baer, "Modified LRU policies for improving second-level cache behavior," in *High-Performance Computer Architecture*, ser. HPCA, 2000, pp. 49–60.

[130] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, ser. MICRO, 2011, pp. 430–441.

[131] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *44th International Symposium on Microarchitecture*, ser. MICRO. ACM, 2011, pp. 430–441.

[132] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "PACMan: prefetch-aware cache management for high performance caching," in *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 442–453.

[133] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 996–1008.

[134] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 449–461.

[135] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 369–383.

[136] V. Young, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *Cache Replacement Championship (CRC'17) held in Conjunction with the International Symposium on Computer Architecture (ISCA)*, 2017.

[137] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016*

49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2016, pp. 1–12.

Vita

Zhan Shi was born in Beijing, China on March 31, 1994. He joined the graduate program in Computer Science at The University of Texas at Austin in August 2017.

Permanent address: 110 Jacob Fontaine Ln, Austin, TX, 78752

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.