# Applying Deep Learning to the Cache Replacement Problem

Zhan Shi
zshi17@cs.utexas.edu
The University of Texas at Austin
Austin, Texas

Xiangru Huang
xrhuang@cs.utexas.edu
The University of Texas at Austin
Austin, Texas

Akanksha Jain
akanksha@cs.utexas.edu
The University of Texas at Austin
Austin, Texas

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

## ABSTRACT

Despite its success in many areas, deep learning is a poor fit for use in hardware predictors because these models are impractically large and slow, but this paper shows how we can use deep learning to help *design* a new cache replacement policy. We first show that for cache replacement, a powerful LSTM learning model can in an offline setting provide better accuracy than current hardware predictors. We then perform analysis to interpret this LSTM model, 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.

The result is the Glider cache replacement policy, which we evaluate on a set of 33 memory-intensive programs from the SPEC 2006, SPEC 2017, and GAP (graph-processing) benchmark suites. In a single-core setting, Glider outperforms top finishers from the $2^{nd}$ Cache Replacement Championship, reducing the miss rate over LRU by 8.9%, compared to reductions of 7.1% for Hawkeye, 6.5% for MPPPB, and 7.5% for SHiP++. On a four-core system, Glider improves IPC over LRU by 14.7%, compared with improvements of 13.6% (Hawkeye), 13.2% (MPPPB), and 11.4% (SHiP++).

## CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; **Architectures**.

## KEYWORDS

caches, cache replacement, deep learning

**ACM Reference Format:**
Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358319

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MICRO-52, October 12–16, 2019, Columbus, OH, USA*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358319

## 1 INTRODUCTION

By extracting useful patterns from large corpuses of data, deep learning has produced dramatic breakthroughs in computer vision, speech recognition, and natural language processing (NLP), and it has shown promise for advancing many other fields of science and engineering [6, 10, 11, 32, 37]. It is therefore natural to wonder if deep learning could drive innovation in the computer architecture domain. In particular, modern microprocessors include many hardware predictors—for branch prediction, for cache replacement, for data prefetching, etc—and deep learning offers a method of training these predictors. Moreover, problems such as cache replacement and data prefetching have been heavily studied over many years, so it makes sense to seek new tools to help us further advance the field.

The term deep learning typically refers to the use of multi-layer neural networks. The use of multiple layers is significant because it provides the ability to learn non-linear relationships at multiple levels of abstraction. The ability to recurse, as seen in recurrent neural networks (RNNs), a type of deep learning model[1], allows RNNs to effectively represent any number of layers, making them quite powerful. And while machine learning models such as RNNs have been extremely successful, powerful new models and variants, such as attention mechanisms [36] and Transformer [53], are constantly being proposed.

Unfortunately, these powerful learning models have not been exploited by architects, who have instead built hardware predictors using simpler learning techniques, such as tables and perceptrons [20, 52]. Perceptrons, however, represent just a single layer of a neural network, so they do not approach the power of deep learning models such as RNNs.

The basic problem is that deep learning models are ill suited for use in hardware predictors. First, deep learning models require enormous resources to train. The training iterates multiple times over the training data and takes hours, days, or even months to complete. Thus, this training is performed offline, which is reasonable for applications such as NLP and computer vision whose prediction targets do not change over time, but offline training is much less effective for hardware predictors because (1) computer programs exhibit time-varying phase behavior , so prediction targets change as the program executes, and (2) the behavior of one program can

[1] A learning model is a broad term that describes any of several different types of machine learning algorithms, such as a neural network, a Support Vector Machine, or a perceptron.

differ wildly from that of another.[2] Second, even after these deep learning models are trained and compressed, they are still too large to be implemented on a chip to make predictions. Finally, these deep models are slow, typically taking several milliseconds to produce a prediction, while hardware predictors typically need a prediction within nanoseconds.

This paper presents a novel approach for addressing this mismatch between deep learning models and hardware predictors, and we demonstrate the strength of our approach by using it to advance the state-of-the-art in the well-studied area of cache replacement.

Our approach uses powerful offline machine learning to develop insights that can improve the design of online hardware predictors. More specifically, our approach has three steps. First, we design a powerful, unconstrained deep RNN model that is trained offline for individual programs. Second, we interpret this offline model to reveal an important insight—described shortly—that is useful for cache designers. Third, we use this insight to design a simple online model that matches the offline model's accuracy with orders of magnitude lower cost. It is this simpler online model that can be implemented in hardware and trained dynamically, much like existing predictors for cache replacement [20, 29, 55].

In choosing an unconstrained offline model, we build on the assumption that control flow is an important aspect of program behavior, so we posit that our offline model should be able to reason about time-ordered events. Thus, to build the unconstrained offline model, we first formulate cache replacement as a sequence labeling problem, where the goal is to assign each element in a sequence of memory accesses a binary label that indicates whether the accessed data should be cached or not (see Figure 2). We use Belady's MIN algorithm to provide oracle labels for our training data. Then, inspired by the recent success of LSTM[3] [16] and attention mechanisms [35, 36] in sequence modeling tasks, we design an attention-based LSTM as our offline model. We find that this offline model improves accuracy significantly (82.6% vs. 72.2% for prior art [20]).

An analysis of the attention weights inside the LSTM then reveals several insights. First, the LSTM benefits when given as input a long history of past load instructions, so we conclude that optimal caching decisions depend on the program's control-flow history and that a long control-flow history is beneficial. Second, the optimal decisions depend primarily on the presence of a few elements in the control-flow history, not the full ordered sequence.

We use these insights to design a new hand-crafted feature that represents a program's control-flow history compactly and that can be used with a much simpler linear learning model known as a support vector machine (SVM). Our SVM is trained online in hardware, and it matches the LSTM's offline accuracy with significantly less overhead; in fact, an online SVM is equivalent to a perceptron, which has been used in commercial branch predictors.

To summarize, this paper makes three contributions:

- We present the first use of deep learning to improve the design of hardware cache replacement policies.
- We design an attention-based LSTM model that (1) significantly improves prediction accuracy and (2) can be interpreted through an attention mechanism to derive important insights about caching.
- We use these insights to produce the Glider[4] cache replacement policy, which uses an SVM-based predictor to outperform the best cache replacement policies from the $2^{nd}$ Cache Replacement Championship. Glider significantly improves upon Hawkeye, the previous state-of-the-art and winner of the $2^{nd}$ Cache Replacement Championship. In a single-core setting, Glider reduces the miss rate over LRU by 8.9%, compared to a reduction of 6.5% for Hawkeye. In a multi-core setting, Glider reduces the miss rate over LRU by 14.7%, compared with 13.6% for Hawkeye.

The remainder of this paper is organized as follows. Section 2 places our work in the context of previous work. Section 3 then describes enough of Hawkeye to make this paper accessible. We then describe our solution and empirically evaluate it in Sections 4 and 5, before presenting concluding remarks.

## 2 RELATED WORK

We now discuss related work in cache replacement before we discuss work that applies machine learning to other parts of the microarchitecture.

### 2.1 Cache Replacement

There has been no prior work that applies deep learning to the hardware cache replacement problem, but replacement policies have evolved from ever more sophisticated heuristic-based solutions to learning-based solutions. Our work continues this trend.

*Heuristic-Driven Solutions.* Most prior cache replacement policies use heuristics that are designed for commonly observed access patterns [2, 7–9, 17, 22, 28, 30, 33, 34, 38, 40, 41, 44, 49–51, 54], leading to variations of the LRU policy, the MRU policy, and combinations of the two. Other heuristics are based on frequency counters [12, 31, 42] or re-reference interval prediction [22]. Still other heuristics estimate the reuse distance of incoming lines, and they protect lines until their expected reuse distance expires [2, 7, 8, 17, 51]. A common drawback of all heuristic-based policies is that they are customized for a limited class of known cache access patterns.

*Learning-Based Solutions.* State-of-the-art solutions [20, 21, 29, 55] take a learning-based approach, as they learn from past caching behavior to predict future caching priorities. Such policies phrase cache replacement as a binary classification problem, where the goal is to predict whether an incoming line is cache-friendly or cache-averse. For example, SDBP [29] and SHiP [55] monitor evictions from a sampler to learn whether a given load instruction is likely to insert cache-friendly lines. Instead of learning the behavior of heuristic-based policies, Hawkeye [20] learns from the optimal solution for past accesses. By providing oracle labels for past accesses, Hawkeye phrases cache replacement as a *supervised learning*

[2] In fact, the behavior of a single program can behave wildly different from one input to another. For example, consider how the behavior of gcc differs when compiling, say, matrix multiplication as opposed to an OS.

[3] LSTM (Long Short Term Memory) is a special variant of recurrent neural networks that can effectively learn long-term dependences in sequential data.

[4] A glider is a simple engineless aircraft that is built from the same principles of flight that underlie more powerful and complex engined aircraft. The glider's simplicity stems from its reliance on more powerful aircraft to do the heavy lifting of getting it to altitude.

problem.[5] Our work builds on Hawkeye, but we use deep learning to design a better predictor than Hawkeye's PC-based predictor.

*Machine Learning-Based Solutions.* There has been little previous work that applies machine learning to the cache replacement problem. The most closely related solutions [27, 52] use an online perceptron [43] to improve the accuracy of cache replacement predictors. In particular, Teran et al. [27] use a perceptron predictor with a short, ordered history of program counters as an input feature. Glider also uses a perceptron with a history of program counters, but it differs from Teran et al.'s solution in its input history representation. In particular, Glider uses an *unordered* history of *unique* PCs, which provides two benefits. First, because it does not include duplicate occurrences of the same PC, Glider uses an effectively longer control-flow history (20 for Glider vs. 3 for perceptron) for the same hardware budget. Second, by relaxing the ordering requirement among these unique PCs, Glider trains much faster than solutions that must learn the behavior of every distinct ordering in different predictor entries.

More recently, Teran et al.'s perceptron was outperformed by MPPPB [27], 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. In Section 5, we show that Glider outperforms MPPPB, but the use of MPPPB's features within a deep neural network model is a promising avenue for future research.

Finally, in earlier work, Jiménez used genetic algorithms [25] to modulate caching priorities for lines that hit in the cache. His work is enlightening because it unifies the notions of cache insertion and promotion, but his solution does not generalize well because it is trained offline on a small number of programs.

## 2.2 Machine Learning in Computer Architecture

Extremely simple machine learning algorithms, such as the perceptron [43], have been used in dynamic branch prediction [23, 24, 26]. For branch prediction, the use of perceptrons enabled the use of long branch histories, which inspired many future academic and commercial branch predictor implementations [45–48].

Complex machine learning algorithms have been adopted directly to implement hardware predictors for shared resource management [5], prefetching [14, 39], and DRAM scheduling [19], but these solutions are largely impractical given their large hardware complexity and large training latencies.

# 3 BACKGROUND

Since our solution uses recurrent neural networks and attention mechanisms, we now provide background on these topics.

## 3.1 The Hawkeye Cache

This paper builds on the Hawkeye cache replacement policy [20], which phrases cache replacement as a supervised learning problem in which a predictor is trained from the optimal caching solution for past cache accesses.

Figure 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 data 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.

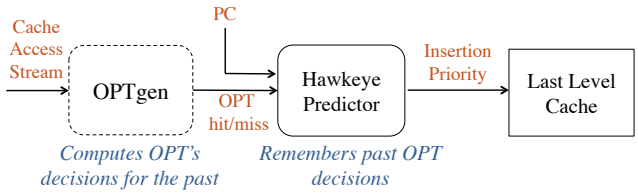



**Figure 1: Overview of the Hawkeye Cache.**

Hawkeye uses the PC as a feature and maintains a table 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.

## 3.2 Recurrent Neural Networks

Recurrent neural networks 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 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 previous hidden state and the current input.

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 4. For example, part-of-speech tagging and name-entity recognition in NLP are both sequence labeling tasks that aim to assign a label to each element of a sentence.

## 3.3 Attention Mechanisms

LSTM has recently been coupled with *attention mechanisms,* which enable a sequence model to focus its attention on certain inputs. For example, when performing machine translation from a source language, say French, to a target language, say English, an attention

[5] Supervised learning is a form of machine learning where the model is learned using labeled data.

mechanism could be used to learn the correlation between words in the original French sentence and its corresponding English translation [36]. 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 [35].

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:

$$a_t(s) = \frac{exp(score(h_t, h_s))}{\sum_{s'} exp(score(h_t, h_{s'}))} \quad (1)$$

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 [36], and the attention weights are further applied to the past hidden states to obtain the context vector:

$$c_t = \sum_s a_{ts} h_s \quad (2)$$

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.

# 4 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.

To improve predictor accuracy, we note that modern replacement policies [20, 29, 55], 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× larger than the typical inputs for LSTM [36], 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 4.1). 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 4.2). 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 4.3). The online version of this SVM model is essentially a perceptron.

## 4.1 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 2).

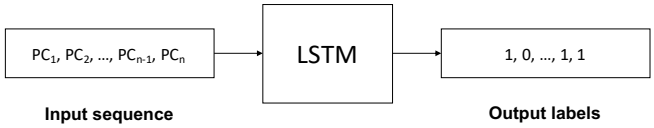



**Figure 2: Our LSTM-based sequence labeling model takes as input a sequence of PCs and produces as output cache-friendly (1) or cache-averse (0) labels.**

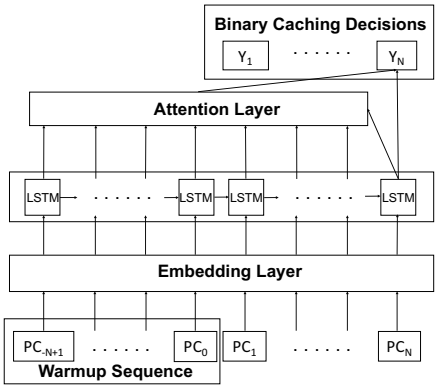



**Figure 3: The attention-based LSTM network architecture. LSTM handles a sequence of input recurrently.**

Figure 3 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 representation for PCs,[6] where the size of the PC vocabulary is the total number of PCs in the

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

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 4.2. Figure 3 shows the attention layer at time step $N$.

Since the memory access trace is too long (see Table 2) for LSTM-based models, we first preprocess the trace by slicing it into fixed-length sequences of 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 5.6 shows our specific hyper-parameters.

## 4.2 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 14). 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 [53] (see Equation 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 paper, we use the dot product as the scoring function.

$$a_t(s) = \frac{exp(f \cdot score(h_t, h_s))}{\sum_{s'} exp(f \cdot score(h_t, h_{s'})} \quad (3)$$

While the attention's scaling factor was originally used to deal with the growing magnitude of dot products with the input dimension [53], 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.

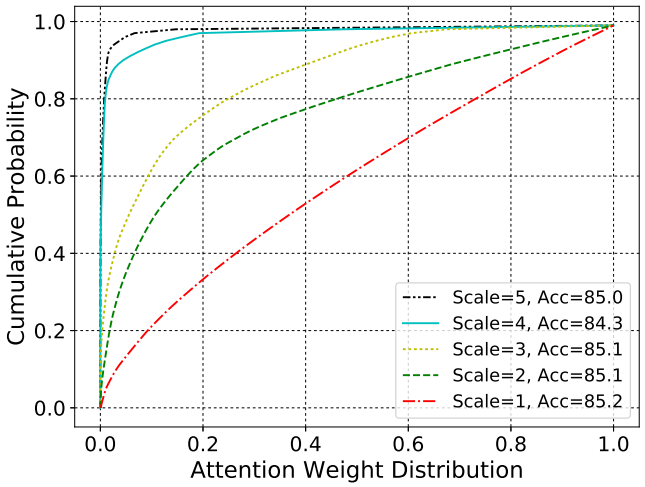
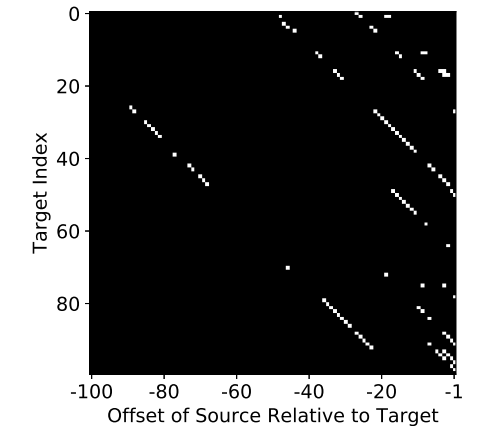


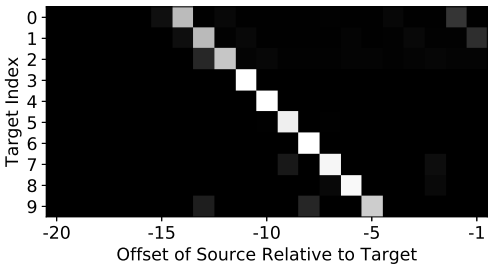

**Figure 4: Cumulative distribution function of attention weight distribution for omnetpp.**

*Insights From The Scaled Attention Layer.* Figure 4 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.

Figures 5(a) and (b) show 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 5(a) shows that typically only a few source memory accesses have dominant attention weights, indicating that only a few source memory accesses influence the caching decision of the target memory access. Zooming in, Figure 5(b) 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. Several




(a) 100 consecutive memory accesses




(b) 10 consecutive memory accesses

**Figure 5: 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.**

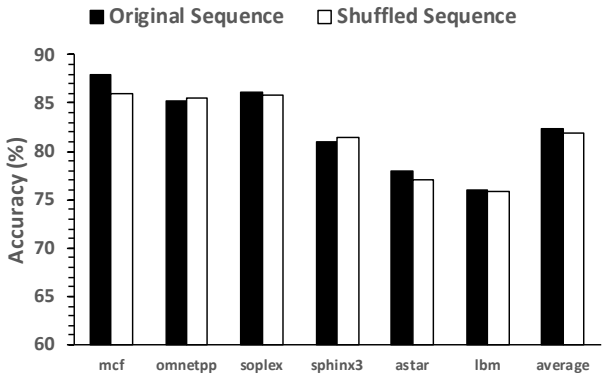



**Figure 6: Accuracy for the original ordered sequence and the randomly shuffled sequence.**

oblique lines can be seen in Figure 5(a), which shows that a few source accesses are important for almost every target access. Therefore, we obtain our second observation:

**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 6 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.*

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 5.5).

## 4.3 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 $t$th 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 7 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

| 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 7: Examples of $k$-sparse binary feature. For simplicity, the total number of PCs is 4 and $k$ is 3.**

$$l(\mathbf{x}, y) = \max(0, 1 - y \cdot \mathbf{w}^T \mathbf{x}) \quad (4)$$

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.*

$$\tilde{l}(\mathbf{x}, y) = \max(0, n - y \cdot \mathbf{w}^T \mathbf{x}) \quad (5)$$

Suppose we are optimizing (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 (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 [52] 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.

## 4.4 Hardware Design

Figure 8 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.

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 8, 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 5.4.

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 [20].

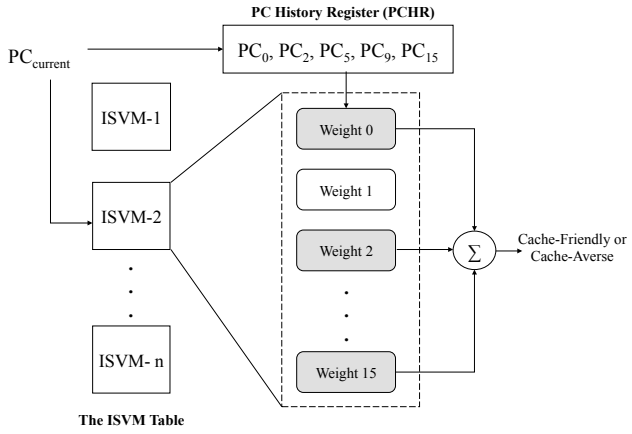



**Figure 8: The Glider predictor.**

*Training.* Glider is trained based on the behavior of a few sampled sets [20, 40]. 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 [27, 52], 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[7]). 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).

# 5 EVALUATION

We evaluate our ideas by comparing in an offline setting our simple ISVM model against a more powerful LSTM model (Section 5.2). We then compare Glider against other online models, ie, against three of the top policies from the $2^{nd}$ Cache Replacement Championship

[7]The Re-Reference Prediction Value (RRPV) counter is used by many modern replacement policies [20, 22, 55] and indicates the relative importance of cache lines.

**Table 1: 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<br>800MHz, 3.2 GB/s for single-core,<br>and 12.8 GB/s for 4-core |

**Table 2: 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 |

(Section 5.3), before discussing the practicality of our solution (Section 5.4).

## 5.1 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 1 shows the parameters for our simulated memory hierarchy.

*Benchmarks.* To evaluate our models, we use the 33 memory-sensitive applications of *SPEC CPU2006* [15], *SPEC CPU2017*, and *GAP* [3], 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 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 [1, 20, 27] 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 [20]. 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 2. 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 [20] and Perceptron [52].

**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.[8]

To evaluate Glider as a practical replacement policy, we compare Glider against **Hawkeye** [20], **SHiP++** [56] and **MPPPB** [27], 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) [4].

## 5.2 Comparison of Offline Models

Figure 9 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

[8] 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 [52].

performance of LSTM. These results confirm our insight that we can approximate the powerful attention-based LSTM with a simpler hardware-friendly predictor.

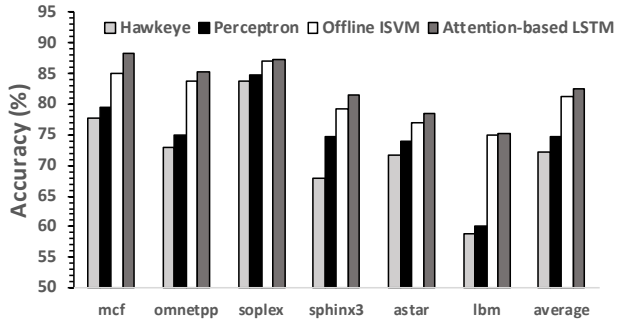



**Figure 9: Accuracy comparison of offline predictors.**

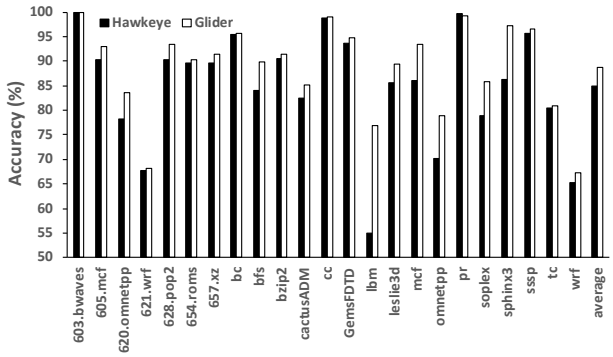



**Figure 10: Accuracy comparison of online predictors.**

## 5.3 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 10 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%. Thus, Glider is as effective as the offline attention-based LSTM model, and insights from offline training carry over to online predictors.

*Single-Core Performance.* Figure 11 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 12 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 13 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.

*Effective Sequence Length.* Figure 14 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 [52]. 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.

## 5.4 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.

*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.)

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

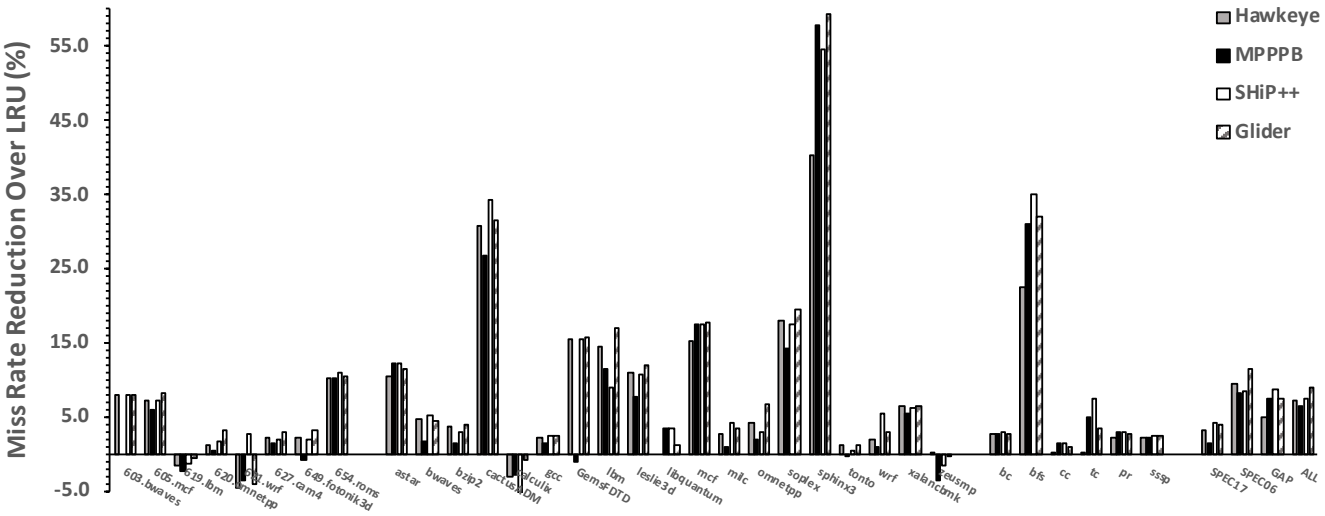



**Figure 11: Miss rate reduction for single-core benchmarks.**

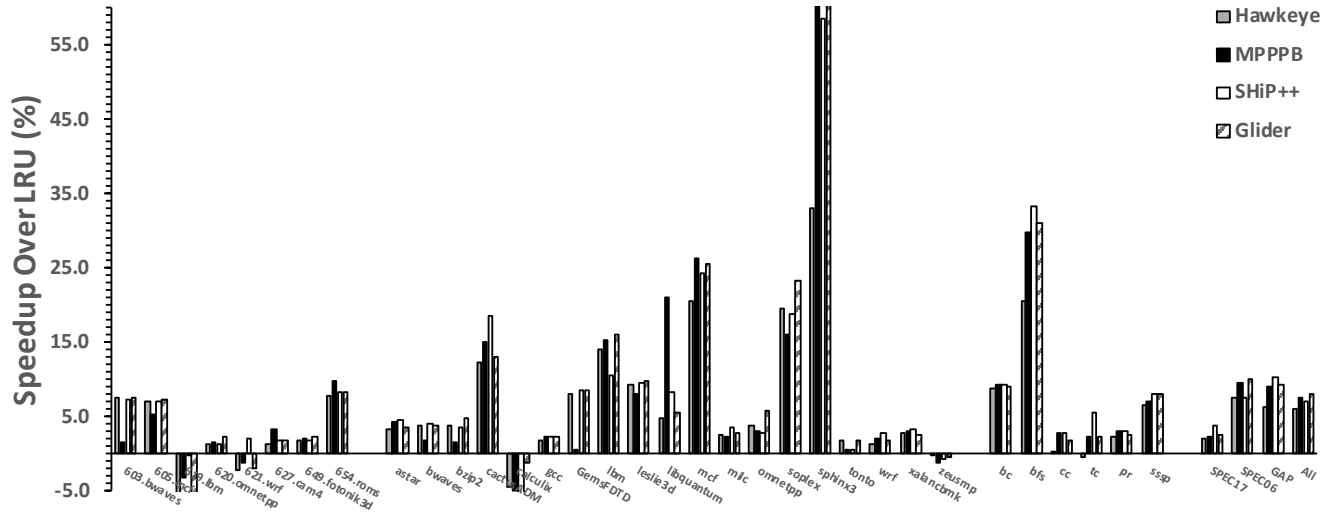



**Figure 12: Speedup comparison for single-core benchmarks.**

**Table 3: 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 |

online manner, that is, by making a single pass over the input data. Figure 15 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 online models, such as, Perceptron and Hawkeye, converge fast but have the limited accuracy.

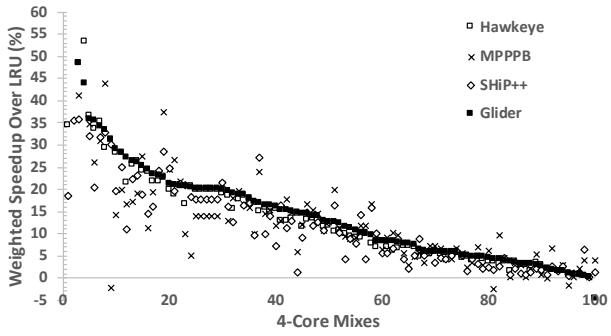



**Figure 13: Weighted speedup for 4 cores with a shared 8MB LLC.**

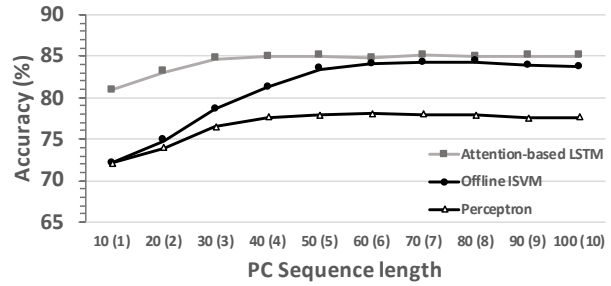



**Figure 14: Sequence length for attention-based LSTM (number of unique PCs for offline ISVM and sequence length for Perceptron).**

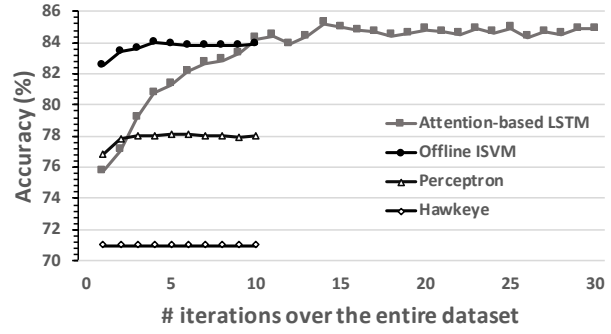



**Figure 15: Convergence of different models.**

*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 [13, 18] 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 15, it's clear that even with further compression techniques, deep learning models are still not ready for direct use in hardware predictors.

## 5.5 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 4: 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 17). Table 4 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).

To understand the accuracy improvement for the target PCs in the *scheduleAt()* method, Figure 16 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, 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.

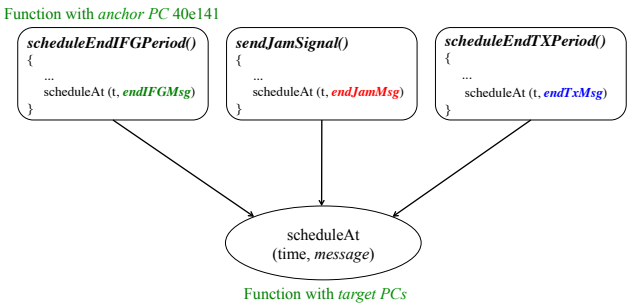



**Figure 16: The anchor PC belongs to one of the calling contexts for the target PCs.**

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

## 5.6 Model Specifications

The hyper-parameters for the attention-based LSTM model and Glider are given in Table 5. 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.

# 6 CONCLUSIONS

In this paper, we have shown how deep learning can help solve the cache replacement problem. We first designed a powerful attention-based LSTM model that uses the control-flow behavior of programs to significantly improve the prediction of caching behavior. While this solution is impractical for hardware cache replacement, we have shown that we can interpret the LSTM model to derive an important insight, and we have used this insight to design an SVM-based predictor that (1) comes close to the accuracy of our LSTM model, (2) is dramatically simpler than our LSTM model, and (3) performs well in an online setting, i.e., as a hardware predictor. The end result is Glider, a practical cache replacement policy that provides accuracy and performance that is superior to the previous state-of-the-art policies.

More broadly, our approach to arrive at Glider suggests that deep learning can play a crucial role in systematically exploring features and feature representations that can improve the effectiveness of much simpler models, such as perceptrons, that hardware designers already use. 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 paper illustrates one instance—cache replacement—where this approach is successful. We hope that the insights and techniques presented in this paper can inspire the design of similar solutions for other microarchitectural prediction problems, such as, branch prediction, data prefetching, and value prediction.

**Table 5: 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 |

# ACKNOWLEDGMENTS

We thank Milad Hashemi, Qixing Huang, Greg Durrett, Philipp Krähenbühl, and Matthew Pabst for their valuable feedback on early drafts. This work was funded in part by a Google Research Award, by NSF Grant CCF-1823546, and by a gift from Intel Corporation through the NSF/Intel Partnership on Foundational Microarchitecture Research.

# REFERENCES

[1] 2017. The 2nd Cache Replacement Championship. http://crc2.ece.tamu.edu/
[2] Jaume Abella, Antonio González, Xavier Vera, and Michael FP O'Boyle. 2005. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)* 2, 1 (2005), 55–77.
[3] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
[4] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* (1966), 78–101.
[5] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *41st International Symposium on Microarchitecture (MICRO)*. 318–329.
[6] Phoebe MR DeVries, Fernanda Viégas, Martin Wattenberg, and Brendan J Meade. 2018. Deep learning of aftershock patterns following large earthquakes. *Nature* 560, 7720 (2018), 632.
[7] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *45th International Symposium on Microarchitecture (MICRO)*. 389–400.

[8] Priyank Faldu and Boris Grot. 2017. Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 180–193.

[9] Hongliang Gao and Chris Wilkerson. 2010. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC 2010-1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*.

[10] Erik Gawehn, Jan A Hiss, and Gisbert Schneider. 2016. Deep learning in drug discovery. *Molecular informatics* 35, 1 (2016), 3–14.

[11] Garrett B Goh, Nathan O Hodas, and Abhinav Vishnu. 2017. Deep learning for computational chemistry. *Journal of computational chemistry* 38, 16 (2017), 1291–1307.

[12] Erik G Hallnor and Steven K Reinhardt. 2000. A fully associative software-managed cache design. In *27th International Symposium on Computer Architecture (ISCA)*. 107–116.

[13] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[14] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *35th International Conference on Machine Learning (ICML)*. 1924–1933.

[15] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* (2006), 1–17.

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

[17] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. 2002. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 209–220.

[18] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[19] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *35th International Symposium on Computer Architecture*. 39–50.

[20] Akanksha Jain and Calvin Lin. 2016. Back to the future: leveraging Belady's algorithm for improved cache replacement. In *43nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 78–89.

[21] Akanksha Jain and Calvin Lin. 2018. Rethinking belady's algorithm to accommodate prefetching. In *45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 110–123.

[22] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA)*. ACM, 60–71.

[23] Daniel A. Jiménez. 2003. Fast Path-Based Neural Branch Prediction. In *36th International Symposium on Microarchitecture (MICRO)*.

[24] Daniel A. Jiménez. 2005. Piecewise Linear Branch Prediction. In *32nd International Symposium on Computer Architecture (ISCA)*.

[25] Daniel A. Jiménez. 2013. Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches. In *46th International Symposium on Microarchitecture (MICRO)*. 284–296.

[26] Daniel A. Jiménez and Calvin Lin. 2001. Dynamic Branch Prediction with Perceptrons. In *7th International Symposium on High Performance Computer Architecture (HPCA)*. 197–206.

[27] Daniel A. Jiménez and Elvira Teran. 2017. Multiperspective Reuse Prediction. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 436–448.

[28] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. 1994. Caching strategies to improve disk system performance. *Computer* 3 (1994), 38–46.

[29] Samira Khan, Yingying Tian, and Daniel A Jimenez. 2010. Sampling Dead Block Prediction For Last-Level Caches. In *43rd International Symposium on Microarchitecture (MICRO)*. 175–186.

[30] Mazen Kharbutli and Yan Solihin. 2005. Counter-Based Cache Replacement Algorithms. In *International Conference on Computer Design (ICCD)*. 61–68.

[31] Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the Least Recently Used and Least Frequently Used policies. *IEEE Trans. Comput.* (2001), 1352–1361.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[33] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 1999. 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. ACM, 134–143.

[34] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *41st International Symposium on Microarchitecture (MICRO)*. 222–233.

[35] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. 2016. Hierarchical question-image co-attention for visual question answering. In *Advances In Neural Information Processing Systems*. 289–297.

[36] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[37] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. 2017. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics* 19, 6 (2017), 1236–1246.

[38] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Record*. ACM, 297–306.

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

[40] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *34th International Symposium on Computer Architecture (ISCA)*. ACM, 381–391.

[41] Moinuddin K Qureshi, Daniel N Lynch, Onur Mutlu, and Yale N Patt. 2006. A case for MLP-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA)*. 167–178.

[42] John T Robinson and Murthy V Devarakonda. 1990. Data cache management using frequency-based replacement. In *the ACM Conference on Measurement and Modeling Computer Systems (SIGMETRICS)*. 134–142.

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

[44] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. 2012. 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*. 355–366.

[45] A Seznec. 2005. Analysis of the O-GEometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 394–405.

[46] André Seznec. 2006. A case for (partially)-tagged geometric history length predictors. *Journal of InstructionLevel Parallelism* (2006).

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

[48] André Seznec. 2011. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 117–127.

[49] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. 1999. EELRU: simple and effective adaptive page replacement. In *ACM SIGMETRICS Performance Evaluation Review*. 122–133.

[50] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H Loh. 2006. Adaptive caches: Effective shaping of cache behavior to workloads. In *39th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 385–396.

[51] Masamichi Takagi and Kei Hiraki. 2004. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 20–30.

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

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[54] Wayne A Wong and Jean-Loup Baer. 2000. Modified LRU policies for improving second-level cache behavior. In *High-Performance Computer Architecture (HPCA)*. 49–60.

[55] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *44th International Symposium on Microarchitecture (MICRO)*. ACM, 430–441.

[56] Vinson Young, Aamer Jaleel, and Moin Qureshi. 2017. SHiP++: Enhancing signature-based hit predictor for improved cache performance. In *Cache Replacement Championship (CRCâĂŹ17) held in Conjunction with the International Symposium on Computer Architecture (ISCA)*.