A Neural Network Prefetcher for Arbitrary Memory Access Patterns

LEEOR PELED, URI WEISER, and YOAV ETSION, Technion-Israel Institute of Technology, Israel

Memory prefetchers are designed to identify and prefetch specific access patterns, including *spatiotemporal* locality (e.g., strides, streams), recurring patterns (e.g., varying strides, temporal correlation), and specific irregular patterns (e.g., pointer chasing, index dereferencing). However, existing prefetchers can only target premeditated patterns and relations they were designed to handle and are unable to capture access patterns in which they do not specialize. In this article, we propose a context-based neural network (NN) prefetcher that dynamically adapts to arbitrary memory access patterns. Leveraging recent advances in machine learning, the proposed NN prefetcher correlates program and machine contextual information with memory accesses patterns, using online-training to identify and dynamically adapt to unique access patterns exhibited by the code. By targeting *semantic locality* in this manner, the prefetcher can discern the useful context attributes and learn to predict previously undetected access patterns, even within noisy memory access streams. We further present an architectural implementation of our NN prefetcher, explore its power, energy, and area limitations, and propose several optimizations. We evaluate the neural network prefetcher over SPEC2006, Graph500, and several microbenchmarks and show that the prefetcher can deliver an average speedup of 21.3% for SPEC2006 (up to 2.3×) and up to 4.4× on kernels over a baseline of PC-based stride prefetcher and 30% for SPEC2006 over a baseline with no prefetching.

CCS Concepts: • **Computing methodologies** → **Neural networks**; Reinforcement learning; • **Computer systems organization** → **Architectures**; • **General and reference** → *Performance*; • **Hardware** → *Semi-conductor memory*;

Additional Key Words and Phrases: Prefetch, neural network, computer architecture

ACM Reference format:

Leeor Peled, Uri Weiser, and Yoav Etsion. 2019. A Neural Network Prefetcher for Arbitrary Memory Access Patterns. *ACM Trans. Archit. Code Optim.* 16, 4, Article 37 (October 2019), 27 pages.
<https://doi.org/10.1145/3345000>

1 INTRODUCTION

Memory prefetching is increasingly critical to processor performance. As a result, most modern processors employ multiple prefetchers that cover a wide range of applications. Existing prefetchers target access patterns that are determined at design time: Classic prefetchers usually target fixed access patterns (e.g., stream, stride [5, 24]), while state-of-the-art prefetchers can capture more flexible variants of spatiotemporal patterns (e.g., variable strides [47, 53], recurring

This research was funded by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and by the Israel Science Foundation (ISF Grant No. 979/17).

Authors' addresses: L. Peled, U. Weiser, and Y. Etsion, Faculty of Electrical Engineering, Technion - Israel Institute of Technology, Haifa 3200003, Israel; emails: leeor@tx.technion.ac.il, uri.weiser@ee.technion.ac.il, yetsion@technion.ac.il.

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 the author(s) 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.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/10-ART37

<https://doi.org/10.1145/3345000>

patterns [26, 41, 56] and temporal correlations [6, 43, 55, 60]). Irregular prefetchers expand the spatiotemporal paradigm by targeting irregular and pointer-based access patterns, but even these typically revolve around known predetermined relations (e.g., pointer chasing [50], double dereferences [61], and structural layout memoization with metadata [28]). Consequently, existing prefetchers cannot dynamically adapt to serve access patterns the designer did not explicitly target, or are unable to handle the abundance of pattern types exhibited by the program.

The resurgence of machine learning gives us new tools to dynamically identify patterns in noisy data streams and capture correlations between data objects. One of the most compelling families of emerging models is that of neural networks (NNs). This family of models offer many variants that can be tuned to different tasks of pattern detection yet do not require prior knowledge of the target patterns. As such, neural networks are not encumbered by our limited view of memory locality artifacts, and are able to detect new forms of relations in existing and emerging applications.

In this article, we propose a neural network prefetcher that dynamically adapts to programs' memory access patterns. The prefetcher uses a variety of workload cues to learn these patterns, ranging from semantic program information [45] to traditional architectural information (e.g., PC, miss history, branch history). The prefetcher employs a small NN implemented using a systolic array. The NN performs buffered inference over program-state context vectors, and back-propagation over their associated prefetch candidates, thereby finding correlation and using it for prefetching.

While neural networks are a powerful tool, integrating them into a memory prefetcher presents many challenges: the learning convergence time must be fast enough, the implementation size must be reasonable, and the energy cost should be justifiable. We examine the constraints of NNs for data prefetching and characterize the key design aspects an NN should capture to effectively serve data prefetching. We also examine the learning speed of the NN and show how fast it can adapt to new access patterns and new program phases.

The contributions of this article are as follows:

- We explore how an on-core online-training neural network can perform context-sensitive prediction tasks such as memory prefetching. We explore different topologies and precision modes of the NN and learn their effect on prediction accuracy and convergence.
- We present the NN-based prefetcher that learns to associate contextual information with future memory accesses and predict them. We propose a physical implementation and show that, while being somewhat aggressive in area, it is feasible in today's technology (taking $\sim 15\text{KB}$ in storage and a $\sim 0.5\text{mm}^2$ matrix multiplier). We show that design can be optimized further using recent NN precision reduction approaches such as quantization [23].
- We evaluate the prefetcher using the gem5 simulator [8] over SPEC2006, Graph500, and a set of kernels. The evaluation shows the prefetcher outperforms a classic stride prefetcher by 21.3% on average and outperforms seven state-of-the-art prefetchers (SMS [56], GHB-PC/DC [43], VLDP [53], IMP [61], BOP [41], Domino [6], and context-RL [45]) by 7.5%–18% on average. Moreover, we show that the proposed prefetcher performs on average as well as having the best of the existing prefetchers for each benchmark (with outliers up to 37% over the best competitor), which suggests the prefetcher exploits all known opportunities, as well as previously undetected ones.

The remainder of this paper is organized as follows: Section 2 describes the challenges of contextual prefetching. Section 3 then describes the neural-network prefetcher, and Section 4 explores design considerations for the neural network. Related work is discussed in Section 5. Section 6 presents our experimental methodology, followed by an analysis of our evaluation in Section 7. Finally, Section 8 concludes the article.

Fig. 1. Access patterns during the main phase of SPEC2006-MCF, over several memory activity ranges. The bottom range (also enlarged in Figure 2) exhibits an interesting pattern that represents a quicksort algorithm. Other patterns include direct linear strides representing the direct fields in the array of *arc* structs, while the scatters represent indirectly linked elements accessed through the pointers there.

2 LEARNING MEMORY ACCESS PATTERNS

Prefetchers observe the sequence of memory accesses performed by the processor and use that information to predict future memory accesses and fetch their associated data ahead of time. The *prefetch* problem statement can therefore be formulated as follows: given the history of memory addresses previously accessed and any complementary information available, produce the most likely *N* addresses to continue the sequence.

When observing a sequence stripped of any additional information, this task is known as the problem of *universal prediction* [40], which is prevalent in the field of information theory and often handled with probabilistic tools. However, since we know that the stream represents accesses to structured data, constructed and used by an algorithm with some form of internal logic and recurrence, we have better chances of finding order in the underlying semantics that produced it. The *semantic locality* paradigm [45] aims to uncover relations between memory accesses that are *consequential* through the use of program context information. The premise behind that is that recurring semantic relations will likely involve similar control flows, specific data values, and spatiotemporal patterns. Thus, tracking the context in which memory accesses occur can help discern their semantic locality.

Figure 1 shows an example of such patterns within multiple sub-streams that are simultaneously active in the MCF benchmark. The different memory streams reflect different data sets (lists, trees, and arrays) that are interleaved by the algorithm. Each stream has different characteristics, some with spatial layout and a linear access pattern, and others with more complicated accesses. Figure 2 isolates one of these streams performing a quicksort phase, going back and forth around a pivot element at each iteration. This mixture of access patterns demonstrates how different access streams interleave to create a single obfuscated access stream.

Fig. 2. Zoom-in on several quicksort phases from Figure 1. The pivot in each step is chosen and elements are rearranged on its sides, creating a cone shape. Each pivot creates a skewed cone according to its location.

Table 1. Example Pattern Types Targeted by Common Prefetchers, Including Streamers/Striders, Spatial-based Prefetchers (SMS), which Repeat a Pattern Around Anchor Addresses, Markov Prefetchers, which Rely on a Probability Model, VLDP, and Delta-Correlation GHB, which Learn Recurring Delta Patterns, and GHB/PC, which Records Patterns Based on PC

Prefetcher	Sample sequence						Sample rule	Context info used
Streamer	A	A+1	A+2	A+3	A+4	A+5	+1,+1,...	Last address
Strider	A	A+n	A+2n	A+3n	A+4n	A+5n	+n,+n,...	Last address
SMS [56]	A	A+n	A+m	B	B+n	B+m	offsets: 0,n,m	Address delta history
Markov [32]	A	B	A	C	A	B	$A \xrightarrow{33\%} B ; A \xrightarrow{66\%} C$	Last address
VLDP [53], GHB */DC [43]	A	A+2	A+3	A+5	A+6	A+8	+2,+1,+2,+1..	Address delta history
GHB PC/* [43]	A_1	B_1	A_2	B_2	A_3	B_3	$PC_A : A_n ; PC_B : B_n$	PC + delta history

Most common prefetchers are designed to handle specific types of semantic relations, most often based on their manifestations as spatiotemporal locality or temporal correlation. Table 1 shows simple examples of such relations. Real applications with complex semantics often exhibit many types of relations (as shown in the MCF example), making the use of predefined relations insufficient. *One of our goals in this article is to explore whether a sufficiently powerful generic learning model can identify and learn all memory access patterns and semantic relations used by a program.*

2.1 Observing Semantic Locality by Context

Most prefetchers employ some portions of the program context to isolate substreams and discover patterns within them. The last column in Table 1 shows examples of the context attributes used by each prefetcher. This raises the question whether truly abundant context will allow better pattern isolation and detection of true semantic locality. The basic premise of context prefetching offers better temporal correlation by using richer semantic information about the memory access stream.

We define a *program context-state* as a vector with the current values of a set of attributes representing the CPU and program state (e.g., register values, branch and access history, access type). Each memory access has a distinct context-state upon dispatch that has conveys semantic meaning regarding the state of the program and therefore correlates with what the program was doing at that exact moment. Even if we cannot *infer* what the program was doing from this information, we can still *correlate* between these actions (including memory accesses) and the state of the program. Strong correlation between contextual states and memory access patterns often indicates true semantic relation [45]. In that sense, context prefetching is an extension of temporal prefetching but it relies on address correlation with a prior context vector rather than just recent memory addresses. Section 3.1 shows the context attributes selected for best semantic correlation.

Fig. 3. NN prefetcher diagram, showing the association queue, the NN unit, the logic for selecting which associations to train, and the prefetch queue that keeps prefetches for feedback.

Fig. 4. 128-bit context state vector used for semantic approximation. LIP history concatenates bits [8:1] of recent load/store LIPs, delta history concatenates bits [14:2] of the access deltas.

Notably, a context that is not detailed enough may fail to expose the desired relations, while an overly detailed context would immediately cause an overfit problem on any temporal correlation recording scheme using queues and pointer tables. An effective context can only be found dynamically using the attributes that are semantically meaningful for a given relation (For example, a certain branch history may indicate that a certain offset of some structure will likely be accessed in the near future, a certain pointer address used by a certain load may indicate the likelihood of another, a data value used for deciding a traversal path will indicate future steps). The task of selectively picking context attributes and identifying their relations to future memory accesses require unique learning capabilities. Thanks to recent developments in the field of machine learning, this can be achieved through online training of neural networks.

3 THE CONTEXTUAL NEURAL NETWORK PREFETCHER

The proposed neural network prefetcher predicts future memory accesses based on current program context. We extract the contextual information and use a neural network to derive prefetch patterns that were associated with similar contexts in the past.

The proposed prefetcher, shown in Figure 3, connects at the L1 cache level and receives the stream of memory addresses accessed by the program and the context state at the time of each access. The context states are represented as bit-vectors describing CPU attributes, as shown in Figure 4. These attributes were already shown [45] to be useful in semantic correlation of access patterns. Accumulated history values are implemented using shift registers.

Fig. 5. The “heart” of the NN prefetcher is a 32×32 systolic array matrix multiplier with 8b precision. The array is fed by various input/error buffers (depending on the phase) and the weight vectors, managed by a controller.

The prefetcher is divided into the following components, shown in Figure 3:

- The *association queue* tracks the history of addresses and context states. The newest context is used to produce a prefetch, while the oldest context is associated with one of the newest addresses to form a pair of context + address for training.
- The *association selector* chooses which of the last few addresses is best associated with the oldest context. This association is only an attempt to find recurring semantic relations—if they prove to be recurring, then the training will strengthen the association representation in the NN. The selector can apply one of several different policies according to their priority. One possible optimization is to split the NN into two parts and apply a different policy for each (effectively training two possible outputs for each input context and comparing their usefulness). The association policies include:
 - The context hash (indexed by folding the context into a 10-bit index using XORs) stores the last associated delta per context. This helps selecting the same recurring association, thereby expediting the convergence of the learning process.
 - The min MSE comparator aims to select the address closest to the current NN output of the training context. We generate an address from the network output for S_n and find the minimal bit-wise distance from the latest d addresses in the association queue to find the most likely recurring association.
 - The max delta selector is suited for stream or recurring groups of accesses where each context may be associated with each access with the same level of recurrence. In that case, we predict the largest delta to make sure the associations are kept at a constant offset and do not mix (which may lead to several contexts predicting the same address, while other addresses remain unassociated.)
- The neural network unit, shown in more detail in Figure 5, learns the associations and produces predictions based on context state vectors.
- The training queue holds pending training tasks scheduled for the NN unit.
- The prefetch queue holds prefetches that were sent (or shadow prefetches with low confidence that were enqueued but not sent), until feedback can be received. If a demand hits

Fig. 6. Schematic workflow of the NN prefetcher given memory access to a_0 under context state S_0 .

Fig. 7. Inference and training steps performed for each miss. The purple arrows are value transfers (weights, expected output), and the gray ones are forward/backward passes through the NN. The blue (left) and green (right) frames represent the overall inference and training phases.

an address on the prefetch queue within a useful depth, then we will schedule a training pass to strengthen that prediction. If a request reaches the end of the queue without being hit by a demand, then we schedule a training pass with the confidence bit set to zero. If an entry is being hit more than once, then we ignore it as it is already associated with an earlier context that fetched it.

Figure 6 illustrates the main workflow of the prefetcher. Figure 7 shows steps related to the NN.

3.1 Prefetcher Workflow

The prefetcher must be able to train on unsupervised samples of context-address pairs based on the observed history. We do not know during training whether the associations will be semantically related. Thus, we rely on recurrence and NN convergence to strengthen true relations over time.

We want to establish a useful prefetching distance to hide the memory latency. We must therefore maintain a long window of history between each context and the address associated with it. This presents the challenge of picking context-address pairs within this window that are likely to be semantically related and are not too close or too far away to render the prefetching ineffective. To address this challenge, we maintain the *association queue*. This queue stores the history of context states and addresses observed on every memory unit access. On every access, we push the current state vector S_0 (described in Figure 4) and the current address at the head of the queue (step 1 in Figure 3). If the access missed the L1, then we also feed S_0 into the neural network, perform an inference phase (step 2), and send the output to be prefetched. Accesses that hit a cache line installed by a prefetch are treated like misses so that we may strengthen the useful association they represent.

The prefetcher is trained to predict address deltas relative to the address of the associated context. This allows bounding the stored values (usually 16 bits are enough), as well as providing generalized predictions on strided patterns. The output is therefore interpreted as one or more delta binary values. Each value is added to the current address A_0 (step 3) to provide a final address that is sent out as a prefetch and enqueued in the prefetch queue for later usefulness feedback.

3.2 Training Associations

The next step is the training phase. We pop the element at the tail of the association queue, which represents the oldest stored context state, S_N , and use the association selector logic to choose one of the recent addresses from the queue to associate with S_N (step 4). Since the addresses are selected from around the head of the queue, they represent a distance of roughly N accesses (where N , the queue size, is selected to fit the desired prefetch depth given the machine average miss time). We used a queue of 128 elements in our runs.

Ideally, we would prefer a fixed distance between the associated context state and address, to better capture recurrence. However, the actual distance of semantically related pairs may change between iterations due to out-of-order execution or code path changes. To mitigate, we allow some variance in the association depth. We achieve that by considering several latest addresses ($A_0 \dots A_d$) as a possible association for S_N , selecting the best candidate for a match using one of several possible policies described in section 3.3.

During the training phase, S_N needs to be fed again into the neural network to reproduce the neuron activation values. However, to avoid a second inference process (the context had already passed that flow when first added to the queue), we record the activation values when the context state first performs its inference. When the context reaches the tail of the association queue, we only need to restore the activations values and inject them into the NN. The NN output is then matched against the recent d addresses read from the queue, and the closest match is found (the address A_i such that the delta from the predicting address ($A_i - A_N$) has the most bits matching the network output: $\arg\min_i(\text{popcnt}((A_i - A_N) \oplus \text{NN}_{out}))$). If the closest match is still too far (MSE is above a threshold of 5), then the min MSE policy will not provide a candidate for association, and we will fall back on the other policies.

The second policy is the context hash. We observed the need to kickstart the initial convergence of the neural network, especially in the presence of conflicting associations (occasions when the same context is associated with different addresses). For that purpose, we add a context hash indexed by a 10-bit XOR-folded hash of the context. Each context vector being associated will store the associated delta into the hash. when selecting a candidate for association, we always prefer addresses that already appear in the context hash for the given context. The context hash has smaller storage lifetime than the neural network (it will be overwritten on either conflicting associations or on overloaded hash values), but it gives us an additional level of confidence that the trained

context/address pair was already observed, and it helps the neural network train faster over recurring associations.

Finally, the third policy is simply selecting the largest delta that is still within reasonable threshold (we use 0x10,000 to indicate there is still potential relation between the accesses). The max-delta policy allows us both the highest prefetching depth (and therefore best latency mitigation), as well as a fixed history-depth, which is important for stream/stride cases where the context is relatively steady and correlates equally with any address at any depth. If we do not keep a fixed history depth for associations in such cases, then we may have redundant associations while other addresses will not be covered at all.

Once an association was selected, the context and the delta are enqueued for training (step 5) and eventually reach the NN unit to perform the backward propagation flow (step 6).

3.3 Parallel Association Policies

As described in the previous section, there are several possible ways of selecting the desired candidate address for association with each context. Different association policies match different types of patterns that the program may exhibit. Therefore, to extract all possible patterns, we can place multiple networks in parallel and train each one independently with the address associated by each policy. Since the inputs to the networks are all identical and some of the hidden neurons may be useful for different networks, we optimize this by dividing only the output layer. This way, different associations are trained in parallel, allowing shared storage and shared learning.

In our experiments, we used one subset range of nodes to train associations with minimal distance in terms of minimal square error (MSE) from the current output of the network. We use the other subset to train the best context-hash match (i.e., the first context-address pair that hits in the context hash). The hash-based associations often give better results during early stages of the run, as they are faster to train, while the slower MSE-based associations improve the overall coverage. Section 7 compares the performance benefit of using one or more association policies in parallel.

The predictions generated by the multiple output subsets are all interpreted as deltas relative to the address of the predicting state. We construct the prefetch addresses and send them to the memory unit. The MSB of each one serves as confidence bit for that prediction. Since triggering prefetches increases overall system bandwidth, the prefetcher may dispatch “*shadow prefetches*,” similar to the RL-context prefetcher [45]. This is done by storing all generated prefetches into a *prefetch queue* and using them to collect feedback, but dispatching only the highest confidence ones to become actual prefetches.

3.4 The NN Prediction Unit

The central component in our prefetcher is the neural network itself, shown in Figure 5. It logically consists of 3 (or more) layers of fully connected *perceptrons*, each performing a linear product between its inputs and a weight vector, and then applying a non-linear activation function on the result. We use ReLU [42] for quantized networks, and a logistic sigmoid [37] for floating-point ones. The depth, layout and precision of the NN are explored in Section 4. Area/power mitigations are discussed in Section 3.9.

The prefetcher performs two steps for each L1 cache miss (triggering on average every ~10 instructions, based on an L1 hit rate of ~80%) described in Figure 7. First, the *prediction* phase performs *inference* over the most recent context state vector (S_0): it feeds the input context vector to the entry layer, computes the neuron activations at each layer, and interprets the values at the output layer as a binary representation of the address delta (relative to the predicting address A_0). The output is rounded to binary values representing distinct bit values, so small inaccuracies during convergence do not affect the value of the outcome. Larger errors can indeed throw the

result off, but they will also affect the square error stronger and facilitate faster convergence, or selection of another association candidate. If the result is distinct enough (the output bits are within valid error thresholds from binary values), then it is sent to the memory unit to be prefetched.

As mentioned in Section 3.3, we then pop the tail of the association queue (the oldest stored state), associate it with one or more of the recent addresses, and train the oldest context to produce the associated delta. The actual training task goes through a training queue, since other sources may also push training tasks (see Section 3.6), but usually the queue may be bypassed. Backward propagation needs to calculate the neurons error gradient based on the activations that existed when the old context was fed to the NN. Since we do not want to pass that context again for inference, we record the activation values (the NN intermediate results, not the network weights) when each context goes through inference (when the context was first pushed to the association queue), so we can inject them back into the NN when that context becomes the oldest and needs to train an association. Once the activation values are restored, we can calculate the normalized error and update the weights.

3.5 NN Systolic Array

During inference, the input vector is fed into a 32 row $\times$ 32 column systolic array. Each column in the array represents an input element, and each row represents a single hidden neuron. Each matrix element multiplies the input value with the corresponding 8-bit floating point weight assigned to it by the neuron. The results are propagated to the next column and accumulated with a 16-bit accumulator per row. The controller then selects the next set of weights and issues another phase of operations for the next set of inputs, while accumulating the partial sums. Since the hidden layer has 32 elements in our design, the neurons can all be computed in parallel, and we require 4 phases to compute the entire input vector. However, thanks to the systolic array, we can start propagating the next phase in a pipelined manner.

Once the hidden layer is calculated, the results pass through an activation unit and are sent back to the input latch for the computation of the output layer. The controller fetches the output weights in parallel. Since there are 32 output neurons and 32 hidden neuron inputs, computing the output neurons requires 4 additional computation phases over the systolic array. The result is activated again and sent back to the main prefetcher unit to construct the address for prefetching. The neuron activation values for S_0 are also stored into the association queue for a later training pass. Overall, the inference phase requires 8 cycles and is therefore performed asynchronously with the stream of memory accesses. If another miss arrives during calculation, then it will be enqueued for a later inference pass. The input context state is already kept in the association queue, so the NN controller simply needs to store a queue of pointers for states pending inference. This holds as long as the overall rate of misses does not exceed the inference bandwidth—if that occurs, then we will randomly drop some of the pending tasks (which is akin to probabilistic sampling, common in some learning paradigms, and shown to be feasible for prefetching [60]).

Subsequently, the prefetcher begins the *training* step by popping the oldest context state, S_n , from the association queue. The neuron values kept (from the time this context state was added to the queue and passed through the inference step) are preloaded into the matrix, and we perform *back-propagation* by comparing them with the desired value (chosen by each of the policies described in Section 3.3 and converted into address deltas relative to A_n). We use gradient-descent to minimize the bitwise min square error (MSE) vector:

$$\delta_{W_{output}} = \frac{\partial MSE}{\partial W_{output}} = \frac{\partial MSE}{\partial output} * \frac{\partial output}{\partial prod} * \frac{\partial prod}{\partial W_{output}}, \quad (1)$$

where *prod* is the vector of pre-activation values, and *output* is the output-layer neuron vector. Since MSE is the squared distance, the derivative is proportional to the delta. Therefore, the first term is a simple subtraction. The second term is the derivative of the activation, which (for ReLU) is a simple step function around 0. The third term is the hidden neuron value. The overall calculation is therefore a 32-element subtraction (which is shared by all trained neurons) followed by an element-wise multiplication for each of the output weight matrix elements, and the result is added to the current weight and is updated in the weight matrix.

After the output weight matrix is updated, the hidden neuron weights are updated in a similar fashion. However, in the hidden layer the impact of each weight affects all output neurons. Therefore, the error gradient of each neuron must incorporate the errors of the entire output layer. This requires a preliminary pass to compute the weighted errors (one FMA per hidden layer weight), followed by another delta calculation phase.

3.6 Prediction Feedback

The neural network must be able to accept feedback for training, both positive and negative. We achieve this by tracking predictions (both real ones and shadow prefetches) in the *prefetch queue*. When a demand hits this queue (step 7 in Figure 3) at a depth deemed useful (we use the same score function as in Reference [45]), we trigger another training pass on the NN to strengthen the context/address pair that was hit. If, however, a prediction is hit outside the useful range, or if it drops off the queue without ever being hit, then negative feedback must be provided. The neural network is not equipped to provide such feedback as there is no way to “untrain” a sample. Instead, we remove a single bit from the NN output range interpreted as predicted delta, and instead train it to hold the confidence of the prediction. Any positive feedback will train it to a high value, and any negative feedback will train it to a low one. Since, unlike the address bits, this output neuron is not intended to be interpreted as a binary value, we are not forced to train it as such. Instead, we can assign gradual values according to the strength of the feedback and achieve more accurate training.

3.7 Predictor Output

The output of the predictor should reflect the address most strongly associated with the input context. We support two possible modes, similar to GHB address-correlation (AC) and delta-correlation (DC) flavors, by training the output to predict:

- Absolute addresses, effectively memorizing the dataset correlations into the neural network. This would require huge storage capabilities, even in the compressed form in which neural networks store information.
- Relative deltas, between the address of the predicting state and the prefetched address. This has two main benefits: It can be bounded, since we assume that associated lines have some degree of locality (and sometimes even enforce that), and it can be used for multiple associations, for example when a strided pattern appears.

One important observation is that the network performs best when the input and output types match, since it is easier to learn a function where the input/output ranges are similar. It is therefore useful to adjust the context to include previous addresses or deltas accordingly.

3.8 Information Encoding

Unlike other learning mechanisms that rely on direct or semi-associative tabular storage, neural networks store their knowledge in the form of node weights. Rather than performing lookups across internal tables, the neural network infers the outcome using multiple paths that combine

the weights of all the active nodes in parallel. Consequently, we do not control how information will be stored in the neural network, and the actual layout depends on the convergence of the training process. A single node in the network might sometimes be associated with a specific, highly specialized feature of the input pattern (e.g., “the grandmother cell” [18], referring to how single neurons can learn to recognize abstract concepts). However, in most cases the neuron will participate in the calculation path of multiple stored elements.

This form of distributed storage increases the number of unique patterns a neural network can learn (i.e., the network’s *expressiveness*) compared to direct tabular representation. For example, a network that learns an *add* function may first attempt to memorize all encountered results, but after sufficiently long training may eventually converge towards a simpler implementation of a bitwise adder simply because memoization will no longer fit in the network’s storage capacity.

Another aspect of the unique NN data encoding is the approximate nature of the results. Even after several rounds of training some recurring association the NN may still output values slightly different due to the time it takes to update the neurons. This process is deliberately slow to allow multiple recurring associations to converge in parallel, hence we only apply a fraction of the error correction as the *learning rate*, and preserve a *momentum* of the prior gradients. Over time, we slowly reduce the learning rate further to protect the accumulated learnings from transient noises.

The square error metrics presented in this article incorporates the average rounding distance required per bit (aggregated across all output bits). Therefore, to minimize it the system seeks to converge as close as possible to round binary values. The gradient descent performed by the NN takes into account the un-rounded output values to a larger error will apply a larger gradient for that output (or hidden) neuron. The square error also affects the association selection so that when matching an address to learn for a given context, we will often pick a candidate that is already close to the network output (which usually best indicates recurrence).

The downside of the binary encoding is that we also lose some properties of the labels such as convexity or locality that may have assisted coverage by allowing predictions over slightly different input contexts. Future work may seek to mitigate that by exploring other output encoding schemes.

Recurrent neural networks (RNNs) represent another important enhancement for neural networks. Normal feed-forward networks connect neurons only between consecutive layers and are effectively stateless with regard to the input stream. RNNs add the notion of loops within the layers, allowing the network to preserve memory of the previous learning steps. Previous inputs take part in the inference of the current input, allowing the network to learn *temporal* functions. This makes such networks effective in learning sequences and patterns (as opposed to a set of unordered samples), since these often represent some forms of temporal relations.

In this work, we focus on a recent type of RNN called *Long Short-Term Memory* (LSTM) [17, 20]. This variant, which adds neurons functioning as “gates” to control the internal loops, is able to learn when the internal node should be used, adjusted, or reset. LSTM allows information to be safely stored for short or long periods and used only when necessary. The information looping functions as a sort of internal memory and may, in some cases, enhance the context visibility beyond what our history allows and up to an arbitrary depth. We add a small number of dedicated LSTM cells (with input, output and forget gates) to examine whether this capability provides better predictability. The LSTM nodes are added on top of the existing network without changing its topology by incorporating them into the last hidden layer with full connectivity to the prior layer and to all output nodes. We also adjust all inference and training steps to include the output according to behavior (i.e., qualifying the output and training with the relevant gates). When a certain pattern is detected the network is able to record it in one or more of the LSTM nodes and cut out further updates, thereby better preserving it.

Fig. 8. Storage required by the NN prefetcher based on size of the hidden layer and the precision bits (assuming an association queue of 128 entries), log-log scaled. The circle shows the implementation used in this article, which takes ~ 15 kB.

3.9 Area and Energy Considerations

Implementing a neural network typically requires a significant die area and consumes a lot of power. This section describes how we mitigate the power and area overheads to achieve a feasible neural network prefetcher. Notably, given the exploratory nature of the NN prefetcher, its power-performance efficiency is lower than in other prefetchers, but given the rapid progress in NN research, this may further improve in the near future.

The first mitigation, often employed in modern neural networks, is trading off precision for power and area. Common NN implementations employ half-precision floating point math (FP-16) [25], and some even reach 8-bit precision [44], with a relatively small impact on the overall accuracy of the learning process. Such a reduction was shown to save area by up to 62% per 2 $\times$ bit reduction [10], due to the number of operations and the simplification of the carry chains. Our implementation therefore uses 8-bit precision for FP calculations.

The fully connected topology of our 3-level neural network will include the following multiply-and-accumulate operations for the feed-forward operation (and approximately the same for the back-propagation step):

$$N_{fma} = N_{input} * N_{hidden} + N_{hidden} * N_{output} \approx 5k \quad (2)$$

Based on estimations by Brunie [10], a single precision FP FMA with fixed point accumulator would require $\sim 2,000 \mu\text{m}^2$ per cell on a 28nm process (adding mixed precision does not seem to add much in area, while improving precision significantly). On a modern 14nm process this should shrink by 4 $\times$, so our 32×32 systolic array would require about $\sim 0.5 \text{mm}^2$ (a small fraction of a modern core). Adding the control logic, data paths and accumulators incurs negligible area overhead compared to the matrix itself. The other significant area consumers are the weights matrix (with an 8-bit weight per FMA operation) and the association queue (with 128 entries of 128-bit state and 8-bit neuron value for each of the $32 + 32$ neurons), resulting in 15kB of storage. Figure 8 shows how the overall storage is affected by the size of the hidden layer and the number of precision bits used.

The next concern is energy consumption. Based on Horowitz et al. [21], with 32bit floating point (single precision) each FMA operation would consume roughly 4.6pJ on a 45nm process, but a 16-bit FP FMA would take only take 1.5pJ, 3 $\times$ less energy. Process scaling provides a significant reduction. Bohr claims [9] a $\sim 1.6 \times$ improvement in energy efficiency per generation on an Intel process; therefore a neural network on 14nm should be $\sim 4 \times$ more energy efficient. We can also assume that reducing the precision from 16-bit to 8-bit would reduce the power and energy by an additional $\sim 3-4 \times$, so the overall energy per step would be ~ 700 pJ.

Finally, recent work shows rapid progress with quantized neural networks, where the values (weights and activation values) are reduced to a few bits [23], with binary neural networks being

Fig. 9. Average square error convergence over time when training the $\sin(x)$ function over different NN depths and sizes (lower is better).

the extreme example [22, 33, 39]. In addition to reducing the area significantly, these techniques make it possible to simplify the calculation steps by orders of magnitude. With a BNN, for example, all FMA operations on binary values are reduced to simple bitwise logic (XNOR and popcount). Higher precision designs also show a significant improvement in area and energy cost. Jouppi et al. [44] quote $6\times$ less energy and area for 8-bit quantized multiplications, as well as $13\times$ less energy and $38\times$ less area for 8-bit quantized additions. The main hurdle with quantized networks is that until recently they were applied only for inference. However, recent work by Courbariaux et al. [12] and by Tang et al. [58] shows promising results in the training domain as well. We adapted some of these techniques in our work to reduce the NN prefetcher overheads even further, but were not yet able to match the performance (as shown in Section 7.4). We therefore select an 8-bit FP precision for our design, but future work may be able to reduce it with a reasonable impact on precision.

4 SEQUENCE PREDICTION WITH NEURAL NETWORKS

Before testing the full prefetcher design on real workloads, we first inspect several variants for training our neural network to observe how well they predict sequences of values. We start with the basic patterns in Table 1 as a benchmark for real memory access streams. We also add several sequences based on functions that represent patterns of various complexities, as a proxy for more complicated access streams: a (shifted) sine function, a polynomial function, a linear line and a pseudo-random function (LFSR based). Each series is fed into the neural network a single value at a time, and the output is trained to provide the next sequential element. Given a series $x_{i=1}^n$, we teach the neural network to predict the values in one of the following association modes: function estimation ($n \rightarrow x_n$), next element prediction ($x_{n-1} \rightarrow x_n$), next element with history ($\{x_{n-2}, x_{n-1}\} \rightarrow x_n$), or delta prediction ($(x_{n-1} - x_{n-2}) \rightarrow (x_n - x_{n-1})$).

We measure the behavior of all benchmark sequences, using the 4 learning modes and 5,000 iteration phases with varying neural network sizes and structures. Results are shown in square error per element (sum over all output bits), averaged over all elements in the sequence.

Figure 9 shows the average square error convergence over time for the same function, during the training process (sum of squared error per bit, averaged across all predicted values on a single cycle). As expected, the fastest converging network is initially the 3-level one, reaching a steady

Fig. 10. Average square error for different sequences and learning modes, on a network with 32 hidden nodes.

state in less than 1,000 iterations. The two 4-level networks converge slower and are much more jittery during that process due to the interplay between the gradients of the two hidden levels, but eventually both surpass the 3-level network. The 5-level neural network fluctuates even more, as expected. However, it does not converge like the shallower networks, even when extending the process to 10^6 iterations, which makes it impractical for online learning.

Figure 10 shows the average square error for the different benchmarks covered in this section. The figure shows the results for a 3-level network with 32 hidden layer neurons, after training over 5,000 iterations (each covering all the elements in the sequence). We observe that different functions benefit from different correlation methods: the polynomial function, for example, benefits from delta learning as it reduces the polynomial degree. SMS benefits from deltas as well, since the pattern is intended to stress such recurring deltas. Markov and VLDP sequences, however, benefit from employing history in the input, since this is the main way to distinguish between the Markov states. LFSR converges to an almost perfect prediction with any input that includes the previous element, due to the simple shift relations between elements (requiring that only the entropy bit be learned). Thus, for best coverage of real sequences, we need to implement multiple modes of correlation and dynamically alternate between them.

Figure 11 plots the rate of convergence over time for all the benchmark functions (using history learning mode as an example), over the first 1,000 steps. The rate of convergence is relatively fast: Most of the patterns had to be replayed only 100–200 times for the network to reach close to the final values, and the error remains stable beyond that point. The only non-converging sequence is the Markov series, due to the order of the benchmark, which traverses each edge on the address “graph” several consecutive times to give it the desired probability before traversing the other edges, thereby making a history length of 1 an unreliable feature.

5 RELATED WORK

5.1 Prefetching Techniques

Falsafi and Wenisch classified [14] prefetchers into:

- **Stream/stride prefetchers** utilize *spatial locality*, common in many applications that use linear data structures placed sequentially in memory. These prefetchers detect the constant stride pattern and run ahead of the demand stream. Most modern CPUs employ flavors of this family. Pugsly et al. [47] proposed the Sandbox prefetcher, which tests different strides before choosing the optimal. Based on a similar concept is the Best-Offset prefetcher by Michaud [41], which won the 2nd Data Prefetching Competition (DPC2) [51]. Another

Fig. 11. Convergence rate for different benchmarking sequences.

participant in DPC2 was the Access Map Pattern Matching (AMPM) prefetcher by Ishii et al. [26], which detects the stride using shifted pattern matching.

- Temporal and address-correlating prefetchers** utilize *temporal locality* between pairs or sequences of accesses, indicating that accesses that appeared with some temporal adjacency in the past will manifest this adjacency in the future as well. The challenge is to isolate the correlated accesses out of a stream of unrelated ones. One of the original examples was the Markov predictor [32]. Later work improved the prefetching depth, including the Global History Buffer Address-Correlation flavors (GHB/AC) by Nesbit and Smith [43], which observe a long history of accesses and isolates recurrences from it (sometimes using the program counter for localization), and the Irregular Stream Buffer (ISB) by Jain and Lin [28], which attempts to restructure the dataset spatially using abundant in-memory meta-data. Later work by Wenisch et al. [60] reduced the meta-data overhead through sampling with the Sampled Temporal Memory Streaming (STMS). More recently, Bakhshalipour et al. proposed an improvement for the temporal lookup mechanism with the Domino prefetcher [6]. We may also consider the trace cache [46] as a form of instruction prefetching that temporally correlates code addresses. Finally, this category can be extended to include works targeting linked data structures such as those by Roth, Moshovos, and Sohi [49, 50], and by Bekerman et al. [7]. These prefetchers track recurring memory accesses and generate jump pointers into irregular data structures.
- Spatially-correlated prefetchers** use an extension of temporal locality that correlates between spatial patterns instead of absolute addresses. These prefetchers seek out recurring spatial patterns (such as deltas and offset patterns) that may repeat locally, such as accesses to the same fields of a structure across different instances. Examples of this family are Spatial Memory Streaming (SMS) by Somogyi et al. [56], the DC flavors of GHB [43], the Variable Length Delta Prefetcher (VLDP) [53] by Shevgoor et al. (predicting deltas based on delta history) and its successor SPP [27] by Kim et al. (which extends the VLDP prefetching range through a recurrence mechanism). Somogyi et al. later also extended SMS to include temporal cues with the Spatio-Temporal Memory Streaming prefetcher [55]. The category also includes irregular prefetchers such as the Indirect Memory Prefetcher (IMP) by

Xiangyao et al. [61], which uses spatial patterns to target array double dereferences of the form $A[B[i]]$.

Semantic prefetchers can be seen as a combination of all the above. The wide context they use allows the correlation learning engine to extract address or delta correlation artifacts, or even other forms of relations between history elements. Such a technique was recently presented by Peled et al. [45], using a *contextual-bandits* scheme and a slew of hardware and software attributes. However, that approach had to rely on complicated context hashing and dimension reduction mechanisms.

5.2 Neural Network-based Predictors

Using neural networks as a means to optimize micro-architectural speculations and predictions has already been proposed. Jiménez and Lin proposed using perceptron-based neural networks for branch prediction [31]. More recently, Teran, Wang, and Jimenez extended this concept for predicting reuse distance and replacement policy [59]. The CPU industry is also incorporating these techniques, with both AMD and Samsung publicly claiming to incorporate neural networks into their branch predictions (in “Ryzen” [2] and in “M1” [3]).

While predicting addresses adds a degree of complexity compared to binary decisions (taken vs. not-taken branch, or keep vs. replace), this work sheds some light on the feasibility of implementing a simple neural network over hardware. Due to the approximated results of neural networks, they seem to fit micro-architectural speculations where mistakes do not have functional effects. However, neural networks are not restricted to that domain. Some researchers also apply them to predict functionally visible results. Esmailzadeh et al. presented a neural network-based predictor for estimating function results [13], used when some degree of approximation is allowed (such as some image processing kernels). Outside the CPU architecture domain, Knoll and Freitas [34] reviewed NNs with *stochastic memoization* for sequence prediction used to optimize compression algorithms.

Siegelmann and Sontag have shown recurrent neural networks (RNN) to be Turing complete, i.e., to have the computational complexity of a Turing machine [54], even with the constraint of having rational weights, thus making it possible to model them with real hardware.

Google teams proposed using neural networks as a Turing-compatible model of computation in some scenarios: Graves, Wayne and Danihelka [15] proposed training such a network to learn the actions of an actual Turing machine, allowing it to learn simple algorithms, while Kurach, Andrychowicz and Sutskever [35] proposed using a neural network with controlled memory nodes (LSTM) to perform complicated algorithms involving linked data structure and array manipulations, focusing on the detection of memory access sequences of specific tasks with small footprints. Recently, Graves et al. published a complete neural network-based compute model called *differentiable neural computer (DNC)* [16], augmenting the neural network with a novel form of internal memory based on parallel weighted read/write matrix operations. DNC was also shown to successfully learn short graph traversals and predict resulting nodes and shortest paths. More recently, Hashemi et al. [19] presented an initial exploration of LSTM neural networks for memory access pattern prediction, relying on PC and address deltas as features. While they have not yet suggested a practical prefetcher, their work offers insights on the potential of deeper networks.

The computational and storage costs of neural networks remain critical limitations. Recent work attempts to mitigate these costs by employing dedicated eDRAM and optical interconnects for storing the weights (DaDianNao by Luo et al. [38]), or by using resistive-memory-based crossbars for storage and analog computation (Shafiee et al. [52]). Other studies seek to simplify the cost of maintaining the NN node weights by reducing them to quantized (e.g., binary or ternary) values

Table 2. Simulator Parameters

Sim. mode	Sys. emulation (SE), accurate timing
CPU	4-core x86 client, OoO core, 4-wide
Queue sizes	192 ROB, 64 IQ, 256 PRF, 32 LQ/SQ
L1 cache	64kB/32kB (Data/Code), 8 ways, 16 MSHRs, 2 cyc.
L2 cache	256kB, 8 ways, 32 MSHRs, 20 cycles
L3 cache	2MB per core, 16 ways, 50 cycles, shared
Main memory	2GB, ~300 cycles access
NN prefetcher	
NN	3–5 layers, 128 inputs, 32 output, 32–128 hidden nodes per layer. 0–16 LSTM nodes
Structures	Assoc. Q.: 128, prefetch Q.: 128, ctxt hash: 1k
Competing prefetchers	
Stride	256 entries, size kB
GHB [43]	GHB size: 256, Hist.: 3, pf. degree: 2, size: 4kB
SMS [56]	PHT size: 2K, AGT size: 32, Filter Table: 32 Regions size: 2kB, size: 20kB
VLDP [53]	DPTs: 3 x 64, 64 OPT, 16 DHB, size 1kB
BOP [41]	256 RRs, 26 offsets, size: ~2kB
IMP [61]	PT: 16, IPD: 4 x 16 columns, size: ~2kB
Domino [6]	4M HT, EIT: 2M x 3 superlines x 3x 64 entries size: ~150MB
ContextRL [45]	CST: 2K x 4 links, Reducer: 16k, prefQ: 128 size: ~30kB

that will be easier to manage in hardware [1, 22]. This approach is mostly used for inference, but quantized training is also explored [23].

6 METHODOLOGY

The neural network-based prefetcher was modeled on the gem5 [8] simulator using system emulation (SE) mode to focus on application user-level code. We used an out-of-order x86 core for realistic behavior. Table 2 specifies the parameters of the simulated system.

A large selection of frameworks is available for implementing neural networks (Caffe [30], torch [11]). However, we preferred instead a simplified in-house model, both due to constraints in simulator integration, as well as to ensure that the implementation is feasible in hardware and has no hidden optimizations. Our NN is a simple fully connected feed-forward network, with optional LSTM nodes added. The network was built to support various floating point precision options, as well as quantized values.

We compare the NN-based prefetcher with seven state-of-the-art prefetchers: VLDP [53], SMS [56], GHB-PC/DC [43], Best-Offset (BOP) [41], IMP [61], Domino [6], and Context-RL [45]. VLDP represents a large family of variable stride prefetchers. SMS represents spatial pattern-based prefetchers. GHB represent simple temporal correlation prefetchers, while its PC/DC flavor is closest to the proposed NN prefetcher as it also focuses on recurring delta correlation and uses some context information. BOP is another spatial variant that optimizes stride prefetching (including variable strides coverage) and is the winner of the 2nd Data Prefetching Competition (DPC2) [51]. IMP is an irregular pattern prefetcher that focuses on some specific access patterns. Domino is the latest temporal correlation prefetcher using massive off-core meta-data, which allows it to

Table 3. Manual Kernels

BFS	Breadth-first search over a spatial (array-based) 10k vertice graph
listsort	Linked list sorted insertion & deletion (over 1M elements)
matmul	Matrix multiplication ($1k \times 512$ by $512 \times 1k$)
suffixArray	Create a suffix array of a 34M long DNA sequence
setCover	Find a greedy approximate minimal set cover over a 20k rMat-based array represented graph
convexHull	Find a convex hull of a 500k elements 2D-Plummer map
spmv	Sparse matrix multiplication
array	Sum a 1M long array
bst	Binary search along a 10k array-based tree
prim	Find a Prim min spanning tree over a linked graph (2^{12} nodes, rMat generated)
list	Pointer chasing along a 200k elements linked list
patterns	Access along a 200k-long array using arithmetically growing index jumps

challenge our claims regarding the NN prefetcher storage capabilities. Context-RL uses a similar context-based prefetching concept as the NN prefetcher but a much simpler and limited learning mechanism, which does not efficiently discern useful context attributes. We also add a simple PC-based stride prefetcher as baseline, as most existing CPUs today implement one [24].

Finally, we use a wide range of common benchmark suites, including SPEC 2006 [57], Graph500 [48], and HPCS [4]. We also add multiple hand-written kernels to achieve high coverage of application behavior. Table 3 describes the manual kernels used. Our evaluation presents all the benchmarks that gain on *any* of the competing prefetchers. SPEC06 average is the geomean over all components (including the ones not gaining).

The benchmarks were compiled with an LLVM v3.6.2 [36]. The simulation was done over 50M instruction phases, sampled at multiple points (at least 50G instructions apart) selected to represent distinct steady-state phases of memory access behavior according to memory workload characterization by Jaleel et al. [29]. For SPEC06 traces, we usually captured 3–4 phases per trace, except when the hotspot did not show benefit for any of the prefetchers checked (usually due to low memory activity).

In this article, we focused on single-threaded performance, which is far more limited by memory latency and is considered harder to optimize even with unconstrained power and area budget. MT runs tend to be more limited by memory bandwidth (since multi-threading by itself is a mitigation for memory latency) and may be less susceptible to prefetching. Threads running on the same core may gain additional performance from shared learning, but that is left for future work.

7 EVALUATION

The neural networks evaluated for our prefetcher are fully connected and have 3 levels (1 hidden layer with 128 neurons), 4 levels (2 hidden layers with 96 and 64 neurons, respectively), and 5 levels

Fig. 12. Speedup comparison of NN-based prefetching vs. other techniques over a baseline with PC-based stride prefetching. Only tests with gain above 5% over any prefetcher are shown.

(3 hidden layers with 96, 64, and 64 neurons). We also ran the 3-level network with an additional 16 LSTM nodes on the last hidden level (without changing the topology otherwise), making it a recurrent NN. Deeper networks are not tested here as even 5-level networks were shown in Section 4 to converge too slowly. Convolutional neural networks (CNN) were also tested but are not shown, since the results were not stable and the partial connectivity we used degraded their performance without exposing any of the desired spatial locality across input nodes.

Figure 12 compares the IPC speedup of the 3-level NN prefetcher against other state-of-the-art prefetchers. The speedup is shown compared to a baseline with a PC-based stride prefetcher (that can learn a single fixed stride per PC). The last column for SPEC06 shows the geomean speedup over a baseline with no prefetching. The workloads shown are the ones that exhibited some minimal degree of sensitivity to prefetching, getting a speedup of at least 5% on any of the tested prefetchers. However, the geomean shown for SPEC06 is over the entire suite, except for the Fortran benchmarks, which could not be compiled on our LLVM/clang, and some benchmarks that had build issues (gcc, perlbench). The graph shows that most of the benchmarks with complex memory access patterns benefit from the neural network-based prefetcher. On average, the NN-prefetcher gains 21.3% (and up to 2.3 $\times$) over the simple stride prefetcher and 30% (and up to 2.7 $\times$) over no prefetching.

Compared to state-of-the-art prefetchers, the NN prefetcher gains 7.5% over GHB PC/DC, 13.5% over SMS, 18% over VLDP, and 13.8% over the RL-based context prefetcher. Moreover, the NN prefetcher provides almost the same average speedup on SPEC06 as selecting the best competitor per trace (with outliers up to 37% over the best competitor), thanks to covering patterns that are not detected by any of the competitors. On kernels, the benefits are even higher, with the max gain

Fig. 13. Breakdown of demand accesses according to prefetch usefulness for the SPEC benchmarks. Useless prefetches are stacked on top of the 100% demands. Each test shows the NN gain vs. the baseline stride prefetcher (S). The secondary y-axis shows the speedup curve.

Fig. 14. Comparison between different association policies for training address selection. *MSE* chooses the address closest to the network output, *Context* policy prefers matches in the context hash, and *Conf* prefers associating addresses of frequent PCs.

(on a linked list traversal test) above 4.4 $\times$. The NN prefetcher performs 5% faster on average (and up to 2.4 $\times$ faster) than selecting the best competitor per trace.

The performance gain of a prefetcher depends on the tradeoffs between useful, useless, and partially useful prefetches. Figure 13 shows the breakdown of demand misses in the L1 cache. Each of them can be categorized as one of the following (sorted by increasing usefulness): a miss that was never prefetched, a prefetch that was triggered but not yet sent (non-timely), or a prefetch that was sent but not yet completed (shorter wait). In addition, we have demands that hit cached lines already hit by previous demands, and useful prefetches (only the first demand to hit them). On top of these categories covering 100% demands, we add bad prefetches: addresses prefetched but never used while in the cache, indicating that the prefetcher was wrong about the address. Speedup is correlated with having many useful prefetches but not too many useless ones. In some cases (e.g., matmul), most of the gain is from reducing the demand miss latency, indicating that the prefetcher is successful, but has potential for further depth tuning.

7.1 Comparing Different NN Schemes

Figure 14 shows the performance of the different association schemes described in Section 3 (over a single phase at 50G instructions skip). The NN is partitioned and trained over different association candidates, producing multiple prefetch candidates. We observed that 4 parallel networks are not

Fig. 15. Comparing NN sizes, depths, and LSTM support.

necessarily better than 2, due to additional thrashing (GemsFDTD, for example, gains less when we activate more networks), and because the PC-based association is less effective (LBM and milc).

Figure 15 shows the impact of network size (number of neurons in the hidden layers), depth, and the use of special features such as LSTM nodes. Interestingly, the differences are very small, and some benchmarks exhibit opposite trends (LBM prefers smaller, simpler networks for its relatively simpler fixed strides, while GemsFDTD and MCF prefer larger networks that can store more temporal correlations for their irregular and linked structures). These negligible differences concur with the observation that deeper networks cannot perform online training fast enough against changing code phases to extract any benefits from their size.

The impact of adding LSTM nodes is generally similar to the impact of increasing the network depth, which indicate that few LSTM nodes may be partially interchangeable with the more expensive addition of whole layers. LBM benefits the most from a simple neural network while MCF, GemsFDTD and astar benefit slightly from adding LSTM nodes. Specifically, we observe that benchmarks using linked data structures (e.g., MCF) or many context-sensitive irregular patterns (e.g., GemsFDTD) cannot memorize their entire history in the NN prefetcher. They can, however, gain from long term memory by storing some critical patterns (e.g., initial list segments, top levels of trees) in the LSTM neurons. LBM, however, has a grid layout that is more spatially recurrent and can train a generic pattern that does not require long term memory.

7.2 Classifying Prefetch Usefulness

Different applications demonstrate different opportunities for the NN prefetcher. First, there are the spatially organized applications, where the program traverses data structures sequentially. In these cases, the neural network prefetcher can learn a constant delta, but so can many of the other prefetchers examined. The neural network prefetcher exhibits higher gains thanks to better context localization (from which the GHB-PC/DC also benefits, albeit to a lesser extent). In some of the cases, the neural network prefetcher also gained thanks to the fixed distance association policy, which improved its coverage and exceeded the prefetching distance of most competitors, thereby covering more of the miss latencies.

The second category is temporally correlated applications such as linked lists. These cases are almost impossible to describe as a function, as they exhibit an almost random dataset layout. The recurrence can, however, be recorded and replayed, at least up to the length that the prefetcher storage can support, and assuming it can avoid history thrashing. In this category (and most notably in the list kernels), the context-RL prefetcher often gains due to the speed of its learning (a single access is enough to generate a complete prediction), and Domino wins when it can fit the entire data set (e.g., BFS). However, on larger data-set sizes (prim uses 2^{12} nodes), the neural network will be able to scale better than the context-RL prefetcher thanks to its distributed storage.

7.3 Comparison with Other Prefetchers

The neural network prefetcher shows substantial speedup on many applications with complicated algorithms and memory access patterns. Irregular applications such as gobmk and zeusmp and kernels such as prim (minimal spanning tree) and spmv (sparse matrix multiplication) show gains thanks to the improved temporal correlation, which allows associative storage of links between addresses. This gain is similar to that of other temporal prefetchers, but allows further depth (thanks to the association queue) and more loose associations (thanks to the approximate nature of the NN), a combination that leads to more opportunities for detecting semantic relations. While PC-localized temporal prefetchers (like GHB-PC/*) will only correlate addresses with the same PC, the NN-prefetch may converge to an association that matches any combination of attributes.

However, some linked data structures such as MCF and the BFS kernel gain less from the NN prefetcher. The variety of possible associations grows due to the wide context used for associations, and is harder to explore thoroughly, therefore the simple correlations in GHB-PC/DC adapt better to MCF. Similarly, Domino (thanks to its huge storage) is able to memorize the BFS graph. Another linked data structure is used in astar, but it does benefit from the NN prefetcher, mostly due to its limited level of connectivity (astar traverses a planar graph). Astar is also able to utilize the full context, attempting to remove any of the attributes caused degradation.

Gobmk presents another irregular workload with a unique gain from NN-prefetch (on the longer skips). This Go benchmark recursively plays game moves down the decision tree and recovers them when they are pruned. This makes the stack of board states big, but the allocation scheme groups related moves spatially. The descent depends on prior decisions, making the branch/PC history an effective heuristic for the chosen paths. The NN usefulness is more apparent on the later phases of the application, when the game stack becomes larger and more fractured.

Notably, NN-prefetcher also shows improved gains on some simple access patterns (matmul, array traversal, LBM) even though competing strided prefetchers should easily cover their strides and learn them much faster than the NN prefetcher. Analysis shows that the NN-prefetcher wins mostly due to association policies that enforce longer lookahead depths that improve the prefetcher timeliness. Strided prefetchers (including the baseline IP-based stride) are often limited in depth. BOP for example allows offsets up to 256 lines (4 pages), and has to test all the offsets in its list along the way to converge on the optimal one in terms of timeliness, assuming one of the strides in that range allows for prefetches to complete in time. Conversely, the NN prefetcher limits the max spatial distance for association at 0x10,000 (just to filter completely unrelated associations), but this value is arbitrary and can be increased for larger data sets.

Finally, one of the closest competitors is context-RL, which, similar to the neural network prefetcher, learns by context/addresses association. Comparing them will therefore illustrate the difference between the storage and complexity limitations (although there may also be minor differences in the policies used by the two methods to select the best associations). For some of the applications, we can see that context-RL provides a higher speedup, the most notable example being the list sort kernel. Analysis shows that the entire list itself is too big to fit in any of the prefetchers, but, since the sorted list is built gradually, a limited head segment of the list that is used during every traversal can be memorized. However, this segment may frequently change while elements are added uniformly across the list. The context-RL prefetcher can immediately update the CST table with better associations (a process that occurs instantly once the score of the new association exceeds the old one). Conversely, the NN prefetcher has to rebuild its weights to reflect the change. Since each of these weights represents multiple elements in parallel this process may break other predictions and take longer to reconverge every time the data structure changes.

Fig. 16. Comparing a network using 64bit FP and 3-bit QNN.

7.4 NN quantization

As described earlier, the underlying assumption in our work is that the neural network can be further optimized in terms of power and area. The leading approach is to reduce the processed values precision through use of quantized neural networks [23], to the extreme of having 1–2 bit values [1, 22, 33]. To evaluate the impact, we implemented a 3-bit version of our NN, albeit it still does not implement state-of-the-art training optimizations [58]. Figure 16 shows a comparison between a 3-layer network (128 hidden nodes) that uses 32-bit FP weights and activations and one that uses 3-bit quantized values (the input and output values are always binary). Most benchmarks sustain only a small hit in their speedup, with the exception of MCF and gemsFDTD that incur significant slowdowns due to their inability to converge with the quantized gradients.

8 CONCLUSIONS

This article presents a neural network memory prefetcher based on *semantic locality*. This recently proposed model argues that locality of reference is an artifact intrinsic to the code semantics and is therefore sensitive to program run-time context. Specific patterns can be correlative with certain context states and these correlations can be detected and classified with a powerful learning model. The proposed NN prefetcher learns the algorithmic properties of programs by feeding machine and program context state elements as inputs to a neural network. The NN is trained at runtime to predict future accesses by correlating context with access patterns. Thanks to the unique online learning abilities of the NN, we can detect locality recurrence through model convergence.

Our main goal in this article is to examine whether the contextual learning capabilities of a neural network are inherently superior to those of other machine learning or heuristic-based techniques. Our analysis demonstrates that this prefetcher outperforms other state-of-the-art prefetchers, providing 7.5% gain over GHB-PC/DC on SPEC06, 13.5% over SMS, 14% over BOP, and 18% over VLDP. Moreover, the NN prefetcher covers a spectrum of spatiotemporal access patterns that can only be handled today by multiple heuristic *spatiotemporal* prefetchers working together.

Notably, our current design is still expensive in terms of area and energy efficiency. As such it may serve well in high-power designs. However, the fast evolution of NN technology will make future networks faster, more compact, and more accurate, making the NN prefetcher or other NN-based predictors more power efficient and open the hatch for new predictor paradigms.

REFERENCES

- [1] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot. 2017. Ternary neural networks for resource-efficient AI applications. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'17)*.
- [2] AMD. 2016. The “Zen” Core Architecture. Retrieved from <http://www.amd.com/en-gb/innovations/software-technologies/zen-cpu>.

- [3] AnandTech. 2016. Hot Chips 2016: Exynos M1 Architecture Disclosed. Retrieved from <http://www.anandtech.com/show/10590/hot-chips-2016-exynos-m1-architecture-disclosed>.
- [4] David A. Bader and Kamesh Madduri. 2005. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the International Conference on High Performance Computing (HiPC'05)*.
- [5] J. L. Baer and T. F. Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing'91)*. ACM, New York, NY, 176–186.
- [6] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *Proceedings of the Symposium on High-Performance Computer Architecture (HPCA'18)*. 131–142. DOI: <https://doi.org/10.1109/HPCA.2018.00021>
- [7] Michael Berman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser. 1999. Correlated load-address predictors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'99)*.
- [8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *Comput. Architect. News* 39, 2 (Aug. 2011).
- [9] Mark Bohr. 2014. 14 nm process technology: Opening new horizons. In *Proceedings of the Intel Developer Forum (IDF'14)*.
- [10] N. Brunie. 2017. Modified fused multiply and add for exact low precision product accumulation. In *Proceedings of the IEEE 24th Symposium on Computer Arithmetic (ARITH'17)*. 106–113. DOI: <https://doi.org/10.1109/ARITH.2017.29>
- [11] Ronan Collobert, Samy Bengio, and Johnny Marthoz. 2002. *Torch: A Modular Machine Learning Software Library*. Technical Report IDIAP Research Report 02-46. Idiap Research Institute.
- [12] Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training deep neural networks with weights and activations constrained to +1 or −1. *CoRR abs/1602.02830* (2016). arxiv:1602.02830 <http://arxiv.org/abs/1602.02830>
- [13] Hadi Esmailzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the International Symposium on Microarchitecture (MICRO'12)*.
- [14] B. Falsafi and T. F. Wenisch. 2014. *Primer on Hardware Prefetching*. Morgan & Claypool.
- [15] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *CoRR abs/1410.5401* (Dec 2014).
- [16] Wayne G. Reynolds, M. Harley, T. Danihelka, I. Grabska-Barwińska, A. Hassabis, and D. Graves. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471–476.
- [17] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. 2015. LSTM: A search space odyssey. *CoRR abs/1503.04069* (2015). <http://arxiv.org/abs/1503.04069>.
- [18] Charles G. Gross. 2002. Genealogy of the ‘grandmother cell’. *The Neuroscientist* 8, 5 (2002), 512–518.
- [19] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. *CoRR abs/1803.02329* (2018). arxiv:1803.02329
- [20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9 (12 1997), 1735–1780.
- [21] M. Horowitz. 2014. 1.1 computing’s energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'14)*. 10–14. DOI: <https://doi.org/10.1109/ISSCC.2014.6757323>
- [22] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, 4107–4115. Retrieved from <http://papers.nips.cc/paper/6573-binarized-neural-networks.pdf>.
- [23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061* (2016).
- [24] Intel. 2018. *Software Optimization Reference Manual (Version 040)*, section 2.5.4. Intel.
- [25] M. H. Ionica and D. Gregg. 2015. The movidius myriad architecture’s potential for scientific computing. *IEEE Micro* 35, 1 (Jan. 2015), 6–14. DOI: <https://doi.org/10.1109/MM.2015.4>
- [26] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM, New York, NY, 499–500.
- [27] P. V. Gratz, A. L. N. Reddy, C. Wilkerson, J. Kim, S. H. Pugsley, and Z. Chishti. 2016. Path confidence-based lookahead prefetching. In *Proceedings of the International Symposium on Microarchitecture (MICRO'16)*. IEEE Press.
- [28] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the International Symposium on Microarchitecture (MICRO'13)*.
- [29] Aamer Jaleel. 2010. Memory characterization of workloads using instrumentation-driven simulation. <http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf>.
- [30] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*.

- [31] D. A. Jiménez and C. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings of the Symposium on High-Performance Computer Architecture (HPCA'01)*.
- [32] D. Joseph and D. Grunwald. 1997. Prefetching using markov predictors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'97)*.
- [33] Minje Kim and Paris Smaragdis. 2016. Bitwise neural networks. *CoRR* (2016). <http://arxiv.org/abs/1601.06071>.
- [34] Byron Knoll and Nando de Freitas. 2012. A machine learning perspective on predictive coding with PAQ8. In *Proceedings of the Data Compression Conference (DCC'12)*. 377–386.
- [35] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. 2016. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*. <http://arxiv.org/abs/1511.06392>.
- [36] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. DOI : <https://doi.org/10.1109/CGO.2004.1281665>
- [37] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 2012. Efficient BackProp. In *Neural Networks: Tricks of the Trade: Second Edition*. Springer, Berlin, Chapter 1, 9–48.
- [38] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen. 2017. DaDianNao: A neural network supercomputer. *IEEE Trans. Comput.* 66, 1 (Jan. 2017).
- [39] J. Redmon, M. Rastegari, V. Ordonez, and A. Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision*. Springer International Publishing, 525–542.
- [40] Neri Merhav and Meir Feder. 1998. Universal prediction. *IEEE Trans. Info. Theory* 44, 6 (1998), 2124–2147.
- [41] P. Michaud. 2016. Best-offset hardware prefetching. In *Proceedings of the Symposium on High-Performance Computer Architecture (HPCA'16)*.
- [42] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*. 807–814.
- [43] Kyle J. Nesbit and James E. Smith. 2004. Data cache prefetching using a global history buffer. In *Proceedings of the Symposium on High-Performance Computer Architecture (HPCA'04)*.
- [44] N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, N. P. Jouppi, C. Young et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*. 1–12.
- [45] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. 285–297.
- [46] Alexander Peleg and Uri Weiser. 1995. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent 5,381,533.
- [47] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. F. Chuang, R. L. Scott, and R. Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *Proceedings of the Symposium on High-Performance Computer Architecture (HPCA'14)*.
- [48] B. W. Barrett, R. C. Murphy, K. B. Wheeler, and J. A. Ang. 2010. *Introducing the Graph 500*. Cray Users Group (CUG).
- [49] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence-based prefetching for linked data structures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*.
- [50] Amir Roth and Gurindar S. Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'99)*.
- [51] C. Wilkerson, S. Pugsley, A. Alameldeen, and H. Kim. 2015. The second data prefetching championship (DPC-2). Retrieved from <http://comparch-conf.gatech.edu/dpc2>.
- [52] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, 14–26.
- [53] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the International Symposium on Microarchitecture (MICRO'15)*.
- [54] Hava T. Siegelmann and Eduardo D. Sontag. 1992. On the computational power of neural nets. In *Proceedings of the Annual Workshop on Computational Learning Theory*. 440–449.
- [55] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal memory streaming. In *Proceedings of the International Symposium on Computer Architecture (ISCA'09)*. 69–80.
- [56] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. In *Proceedings of the International Symposium on Computer Architecture (ISCA'06)*.

- [57] Standard Performance Evaluation Corporation. [n.d.]. SPEC2006. Retrieved from <http://www.spec.org>.
- [58] Wei Tang, Gang Hua, and Liang Wang. 2017. How to train a compact binary neural network with high accuracy? In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*.
- [59] Elvira Teran, Zhe Wang, and Daniel A. Jiménez. 2016. Perceptron learning for reuse prediction. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–12.
- [60] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *Proceedings of the Symposium on High-Performance Computer Architecture (HPCA'09)*. 79–90.
- [61] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 178–190.

Received January 2019; revised July 2019; accepted July 2019