Learning Memory Access Patterns

Milad Hashemi [1] **Kevin Swersky** [1] **Jamie A. Smith** [1] **Grant Ayers** [2 *] **Heiner Litz** [3 *] **Jichuan Chang** [1]
Christos Kozyrakis [2] **Parthasarathy Ranganathan** [1]

Abstract

The explosion in workload complexity and the recent slow-down in Moore's law scaling call for new approaches towards efficient computing. Researchers are now beginning to use recent advances in machine learning in software optimizations, augmenting or replacing traditional heuristics and data structures. However, the space of machine learning for computer hardware architecture is only lightly explored. In this paper, we demonstrate the potential of deep learning to address the von Neumann bottleneck of memory performance. We focus on the critical problem of learning memory access patterns, with the goal of constructing accurate and efficient memory prefetchers. We relate contemporary prefetching strategies to n-gram models in natural language processing, and show how recurrent neural networks can serve as a drop-in replacement. On a suite of challenging benchmark datasets, we find that neural networks consistently demonstrate superior performance in terms of precision and recall. This work represents the first step towards practical neural-network based prefetching, and opens a wide range of exciting directions for machine learning in computer architecture research.

1. Introduction

The proliferation of machine learning, and more recently deep learning, in real-world applications has been made possible by an exponential increase in compute capabilities, largely driven by advancements in hardware design. To maximize the effectiveness of a given design, computer architecture often involves the use of prediction and heuristics. Prefetching is a canonical example of this, where instructions or data are brought into much faster storage well in advance of their required usage.

Prefetching addresses a critical bottleneck in von Neumann computers: computation is orders of magnitude faster than accessing memory. This problem is known as the memory wall (Wulf & McKee, 1995), and modern applications can spend over 50% of all compute cycles waiting for data to arrive from memory (Kozyrakis et al., 2010; Ferdman et al., 2012; Kanev et al., 2015). To mitigate the memory wall, microprocessors use a hierarchical memory system, with small and fast memory close to the processor (i.e., caches), and large yet slower memory farther away. Prefetchers predict when to fetch what data into cache to reduce memory latency, and the key towards effective prefetching is to attack the difficult problem of predicting memory access patterns.

Predictive optimization such as prefetching is one form of speculation. Modern microprocessors leverage numerous types of predictive structures to issue speculative requests with the aim of increasing performance. Historically, most predictors in hardware are table-based. That is, future events are expected to correlate with past history tracked in lookup tables (implemented as memory arrays). These memory arrays are sized based on the working set, or amount of information that the application actively uses. However, the working sets of modern datacenter workloads are orders of magnitude larger than those of traditional workloads such as *SPEC CPU2006* and continue to grow (Ayers et al., 2018; Ferdman et al., 2012; Gutierrez et al., 2011; Hashemi et al., 2016). This trend poses a significant challenge, as prediction accuracy drops sharply when the working set is larger than the predictive table. Scaling predictive tables with fast-growing working sets is difficult and costly for hardware implementation.

Neural networks have emerged as a powerful technique to address sequence prediction problems, such as those found in natural language processing (NLP) and text understanding (Bengio et al., 2003; Mikolov et al., 2010; 2013). Simple perceptrons have even been deployed in hardware (e.g., SPARC T4 processor (Golla & Jordan, 2011)), to handle branch prediction (Jiménez & Lin, 2001). Yet, exploring the effectiveness of sequential learning algorithms in microarchitectural designs is still an open area of research.

[*]Equal contribution [1]Google [2]Stanford University [3]University of California, Santa Cruz. Correspondence to: Milad Hashemi <miladh@google.com>, Kevin Swersky <kswersky@google.com>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

In this paper, we explore the utility of sequence-based neural networks in microarchitectural systems. In particular, given the challenge of the memory wall, we apply sequence learning to the difficult problem of prefetching.

Prefetching is fundamentally a regression problem. The output space, however, is both vast and extremely sparse, making it a poor fit for standard regression models. We take inspiration from recent works in image and audio generation that discretize the space, namely PixelRNN and Wavenet (Oord et al., 2016a;b). Discretization makes prefetching more analogous to neural language models, and we leverage it as a starting point for building neural prefetchers. We find that we can successfully model the output space to a degree of accuracy that makes neural prefetching a very distinct possibility. On a number of benchmark datasets, we find that recurrent neural networks significantly outperform the state-of-the-art of traditional hardware prefetchers. We also find that our results are interpretable. Given a memory access trace, we show that the RNN is able to discern semantic information about the underlying application.

2. Background

2.1. Microarchitectural Data Prefetchers

Prefetchers are hardware structures that predict future memory accesses from past history. They can largely be separated into two categories: stride prefetchers and correlation prefetchers. Stride prefetchers are commonly implemented in modern processors and lock onto stable, repeatable deltas (differences between subsequent memory addresses) (Gindele, 1977; Jouppi, 1990; Palacharla & Kessler, 1994). For example, given an access pattern that adds four to a memory address every time (0, 4, 8, 12), a stride prefetcher will learn that delta and try to prefetch ahead of the demand stream, launching parallel accesses to potential future address targets (16, 20, 24) up to a set prefetch distance.

Correlation prefetchers try to learn patterns that may repeat, but are not as consistent as a single stable delta (Charney & Reeves, 1995; Lai et al., 2001; Somogyi et al., 2006; Roth et al., 1998). They store the past history of memory accesses in large tables and are better at predicting more irregular patterns than stride prefetchers. Examples of correlation prefetchers include Markov prefetchers (Joseph & Grunwald, 1997), GHB prefetchers (Nesbit & Smith, 2004), and more recent work that utilizes larger in-memory structures (Jain & Lin, 2013). Correlation prefetchers require large, costly tables, and are typically not implemented in modern multi-core processors.

2.2. Recurrent Neural Networks

Deep learning has become the model-class of choice for many sequential prediction problems. Notably, speech recognition (Hinton et al., 2012) and natural language processing (Mikolov et al., 2010). In particular, RNNs are a preferred choice for their ability to model long-range dependencies. LSTMs (Hochreiter & Schmidhuber, 1997) have emerged as a popular RNN variant that deals with training issues in standard RNNs, by propagating the internal state additively instead of multiplicatively. An LSTM is composed of a hidden state $\mathbf{h}$ and a cell state $\mathbf{c}$, along with input $\mathbf{i}$, forget $\mathbf{f}$, and output gates $\mathbf{o}$ that dictate what information gets stored and propagated to the next timestep. At timestep N, input $\mathbf{x}_N$ is presented to the LSTM, and the LSTM states are computed using the following process:

1. Compute the input, forget, and output gates

$$\mathbf{i}_N = \sigma(W_i[\mathbf{x}_N, \mathbf{h}_{N-1}] + \mathbf{b}_i)$$
$$\mathbf{f}_N = \sigma(W_f[\mathbf{x}_N, \mathbf{h}_{N-1}] + \mathbf{b}_f)$$
$$\mathbf{o}_N = \sigma(W_o[\mathbf{x}_N, \mathbf{h}_{N-1}] + \mathbf{b}_o)$$

2. Update the cell state

$$\mathbf{c}_N = \mathbf{f}_N \odot \mathbf{c}_{N-1} + \mathbf{i}_N \odot \tanh(W_c[\mathbf{x}_N, \mathbf{h}_{N-1}] + \mathbf{b}_c)$$

3. Compute the LSTM hidden (output) state

$$\mathbf{h}_N = \mathbf{o}_N \odot \tanh(\mathbf{c}_N)$$

Where $[\mathbf{x}_N, \mathbf{h}_{N-1}]$ represents the concatenation of the current input and previous hidden state, $\odot$ represents element-wise multiplication, and $\sigma(u) = \frac{1}{1+\exp(-u)}$ is the sigmoid non-linearity.

The above process forms a single LSTM layer, where $W_{\{i,f,o,c\}}$ are the weights of the layer, and $\mathbf{b}_{\{i,f,o,c\}}$ are the biases. LSTM layers can be further stacked so that the output of one LSTM layer at time N becomes the input to another LSTM layer at time N, allowing for greater modeling flexibility with relatively few extra parameters.

3. Problem Formulation

3.1. Prefetching as a Prediction Problem

Prefetching is the process of predicting future memory accesses that will miss in the on-chip cache and access memory based on past history. Each of these memory addresses are generated by a memory instruction (a load/store). Memory instructions are a subset of all instructions that interact with the addressable memory of the computer system.

Many hardware proposals use two features to make these prefetching decisions: the sequence of caches miss addresses that have been observed so far and the sequence of instruction addresses, also known as program counters (PCs), that are associated with the instruction that generated each of the cache miss addresses.

PCs are unique tags, that is each PC is unique to a particular instruction that has been compiled from a particular function in a particular code file. PC sequences can inform the model of patterns in the control flow of higher level code, while the miss address sequence informs the model of which address to prefetch next. In modern computer systems, both of these features are represented as 64-bit integers.

Therefore, an initial model could use two input features at a given timestep N. It could use the address and PC that generated a cache miss at that timestep to predict the address of the miss at timestep $N + 1$.

However, one concern quickly becomes apparent: the address space of an application is extremely sparse. In our training data with O(100M) cache misses, only O(10M) unique cache block miss addresses appear on average out of the entire 2^{64} physical address space. This is displayed when we plot an example trace from *omnetpp* (a benchmark from the standard *SPEC CPU2006* benchmark suite (Sta, 2006)) in Figure 1, where the red datapoints are cache miss addresses[1]. The wide range and severely multi-modal nature of this space makes it a challenge for time-series regression models. For example, neural networks tend to work best with normalized inputs, however when normalizing this data, the finite precision floating-point representation results in a significant loss of information. This issue affects modeling at both the input and output levels, and we will describe several approaches to deal with both aspects.

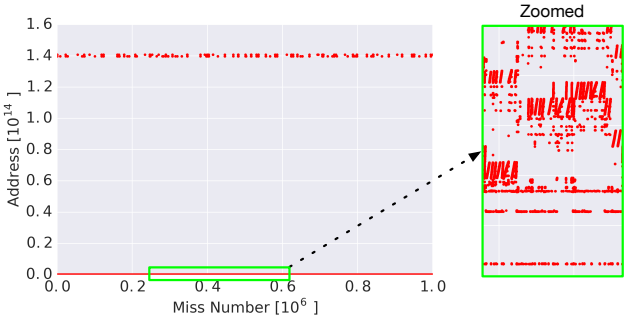


Figure 1. Cache miss addresses on the *omnetpp* dataset, demonstrating sparse access patterns at multiple scales.

3.2. Prefetching as Classification

Rather than treating the prefetching problem as regression, we opt to treat the address space as a large, discrete vocabulary, and perform classification. This is analogous to next-word or character prediction in natural language processing. The extreme sparsity of the space, and the fact that some addresses are much more commonly accessed than others, means that the effective vocabulary size can actually be manageable for RNN models. Additionally, the model gains flexibility by being able to produce multi-modal outputs, compared to unimodal regression techniques that assume e.g., a Gaussian likelihood.

However, there are 2^{64} possible softmax targets, so a quantization scheme is necessary. Importantly, in order to be useful, a prefetch must be within a cache line to be completely accurate, usually within 64 bytes. There is a second order benefit if it is within a page, usually 4096 bytes, but even predicting at the page level would leave 2^{52} possible targets. In (Oord et al., 2016b), they predict 16-bit integer values from an acoustic signal. To avoid having to apply a softmax over 2^{16} values, they apply a non-linear quantization scheme to reduce the space to 256 categories. This form of quantization is inappropriate for our purposes, as it decreases the resolution of addresses towards the extremes of the address space, whereas in prefetching we need high resolution in every area where addresses are used.

Luckily, programs tend to behave in predictable ways, so only a relatively small (but still large in absolute numbers), and consistent set of addresses are ever seen. Our primary quantization scheme is to therefore create a vocabulary of common addresses during training, and to use this as the set of targets during testing. This reduces the coverage, as there may be addresses at test time that are not seen during training time, however we will show that for reasonably-sized vocabularies, we capture a significant proportion of the space. The second approach we explore is to cluster the addresses using clustering on the address space. This is akin to an adaptive form of non-linear quantization.

Due to dynamic side-effects such as address space layout randomization (ASLR), different runs of the same program will lead to different raw address accesses (PaX Team, 2003). However, given a layout, the program will behave in a consistent manner. Therefore, one potential strategy is to predict deltas, $\Delta_N = \text{Addr}_{N+1} - \text{Addr}_N$, instead of addresses directly. These will remain consistent across program executions, and come with the benefit that the number of uniquely occurring deltas is often orders of magnitude smaller than uniquely occurring addresses. This is shown in Table 1, where we show the number of unique PCs, addresses, and deltas across a suite of program trace datasets. We also show the number of unique addresses and deltas required to achieve 50% coverage. In almost all cases, this is much smaller when considering deltas. In our models, we therefore use deltas as inputs instead of raw addresses.

4. Models

In this section we introduce two LSTM-based prefetching models. The first version is analogous to a standard language model, while the second exploits the structure of the

[1]Cache miss addresses are from a three level simulated cache hierarchy with a 32 KB L1, 256 KB L2, and 1.25 MB Last Level Cache (LLC), similar to a single thread context from an Intel Broadwell microprocessor.

Table 1. Program trace dataset statistics. M stands for million.

Dataset	# Misses	# PC	# Addrs	# Deltas	# Addrs 50% mass	# Deltas 50% mass
gems	500M	3278	13.11M	2.47M	4.28M	18
astar	500M	211	0.53M	1.77M	0.06M	15
bwaves	491M	893	14.20M	3.67M	3.03M	2
lbm	500M	55	6.60M	709	3.06M	9
leslie3d	500M	2554	1.23M	0.03M	0.23M	15
libquantum	470M	46	0.52M	30	0.26M	1
mcf	500M	174	27.41M	30.82M	0.07M	0.09M
milc	500M	898	3.74M	9.68M	0.87M	46
omnetpp	449M	976	0.71M	5.01M	0.12M	4613
soplex	500M	1218	3.49M	5.27M	1.04M	10
sphinx	283M	693	0.21M	0.37M	0.03M	3
websearch	500M	54600	77.76M	96.41M	0.33M	5186

memory access space in order to reduce the vocabulary size and reduce the model memory footprint.

4.1. Embedding LSTM

Suppose we restricted the output vocabulary size in order to only model the most frequently occurring deltas. According to Table 1, the size of the vocabulary required in order to obtain at best 50% accuracy is usually O(1000) or less, well within the capabilities of standard language models. Our first model therefore restricts the output vocabulary size to 50,000 of the most frequent, unique deltas. For the input vocabulary, we include all deltas as long as they appear in the dataset at least 10 times. Expanding the vocabulary beyond this is challenging, both computationally and statistically. We leave an exploration of approaches like the hierarchical softmax (Mnih & Hinton, 2009) to future work.

We refer to this model as the embedding LSTM, as illustrated in Figure 2. It uses a categorical (one-hot) representation for both the input and output deltas. At timestep N, the input PC_N and Δ_N are individually embedded and then the embeddings are concatenated and fed as inputs to a two-layer LSTM. The LSTM then performs classification over the delta vocabulary, and the K highest-probability deltas are chosen for prefetching [2].

In a practical implementation, a prefetcher can return several predictions. This creates a trade-off, where more predictions increases the probability of a cache hit at the next timestep, but potentially removes other useful items from the cache. We opt to prefetch the top-10 predictions of the LSTM at each timestep. Other possibilities that we do not explore here include using a beam-search to predict the next n deltas, or to learn to directly predict N to $N+n$ steps ahead in one forward pass of the LSTM.

There are several limitations to this approach. First, a large vocabulary increases the model's computational and storage footprint. Second, truncating the vocabulary necessarily puts a ceiling on the accuracy of the model. Finally, dealing with rarely occurring deltas is non-trivial. This is analogous to the rare word problem in NLP (Luong et al., 2015).

[2]Directly predicting probabilites is another advantage that classification provides over traditional hardware.

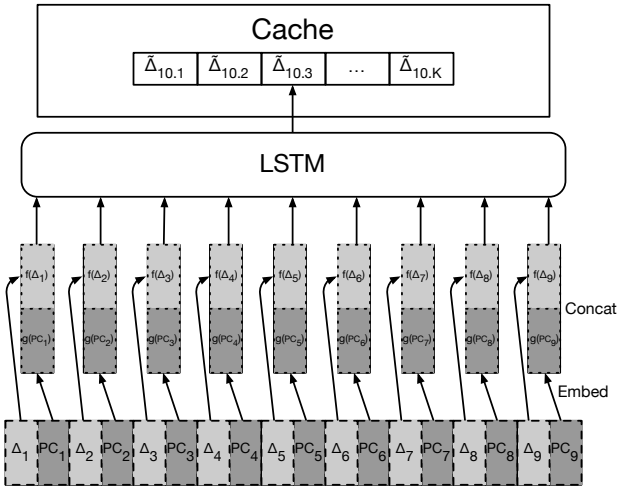


Figure 2. The embedding LSTM model. f and g represent embedding functions.

4.2. Clustering + LSTM

We hypothesize that much of the interesting interaction between addresses occurs locally in address space. As one example, data structures like structs and arrays tend to be stored in contiguous blocks, and accessed repeatedly. In this model, we exploit this idea to design a prefetcher that very carefully models *local* context, whereas the embedding LSTM models both local and global context.

By looking at narrower regions of the address space, we can see that there is indeed rich local context. We took the set of addresses from *omnetpp* and clustered them into 6 different regions using k-means. We show two of the clusters in Figure 3, and the rest can be found in the appendix.

To assess the relative accuracy of modeling local address-space regions, we first cluster the raw address space using k-means. The data is then partitioned into these clusters, and deltas are computed **within** each cluster. A visual example

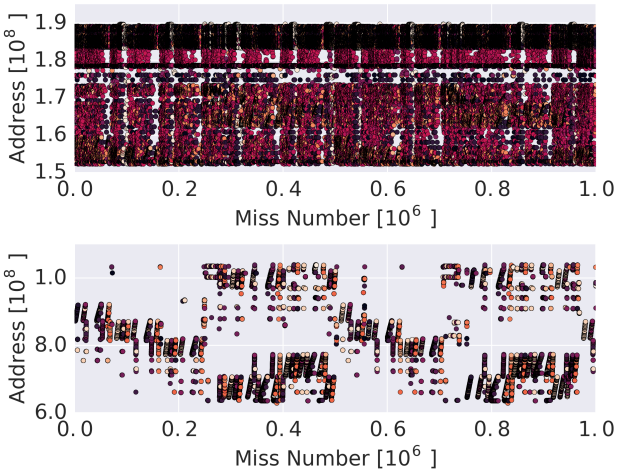


Figure 3. Two of six k-means clusters from *omnetpp*. Memory accesses are colored according to the PC that generated them.

of this is shown in Figure 4a. We found that one of the major advantages of this approach is that the set of deltas within a cluster is significantly smaller than the global vocabulary, alleviating some of the issues with the embedding LSTM.

To reduce the size of the model, we use a multi-task LSTM to model all of the clusters. Stated another way, we use an LSTM to model each cluster independently, but tie the weights of the LSTMs. However, we provide the *cluster ID* as an additional feature, which effectively gives each LSTM a different set of biases.

The partitioning of the address space into narrower regions also means that the set of addresses within each cluster will take on roughly the same order of magnitude, meaning that the resulting deltas can be effectively normalized and used as real-valued inputs to the LSTM. This allows us to further reduce the size of the model, as we do not need to keep around a large matrix of embeddings. Importantly, we still treat next-delta prediction as a classification problem, as we found that regression is still too inaccurate to be practical [3].

This version of the LSTM addresses some of the issues of the embedding LSTM. The trade-offs are that it requires an additional step of pre-processing to cluster the address space, and that it only models local context. That is, it cannot model the dynamics that cause the program to access different regions of the address space.

5. Experiments

A necessary condition for neural networks to be effective prefetchers is that they must be able to accurately predict cache misses. Our experiments measure their effectiveness in this task when compared with traditional hardware.

[3]The reason for this is after *de-normalization*, small inaccuracies become dramatically magnified.

5.1. Data Collection

The data used in our evaluation is a dynamic trace that contains the sequence of memory addresses that an application computes. This trace is captured by using a dynamic instrumentation tool, Pin (Luk et al., 2005), that attaches to the process and emits a "PC, Virtual Address" tuple into a file every time the instrumented application accesses memory (every load or store instruction).

This raw access trace mostly contains accesses that hit in the cache (such as stack accesses, which are present in the data cache). Since we are focused on predicting cache misses, we obtain the sequence of cache misses by simulating this trace through a simple cache simulator that emulates an Intel Broadwell microprocessor (Section 3.1).

To evaluate our proposals, we use the memory intensive applications of SPEC CPU2006. This is a standard benchmark suite that is used pervasively to evaluate the performance of computer systems. However, SPEC CPU2006 also has small working sets when compared to modern datacenter workloads. Therefore in addition to SPEC benchmarks, we also include Google's *websearch* workload. *Websearch* is a unique application with complex access patterns that exemplifies enterprise-scale software development and drives industrial hardware platforms.

5.2. Experimental Setup

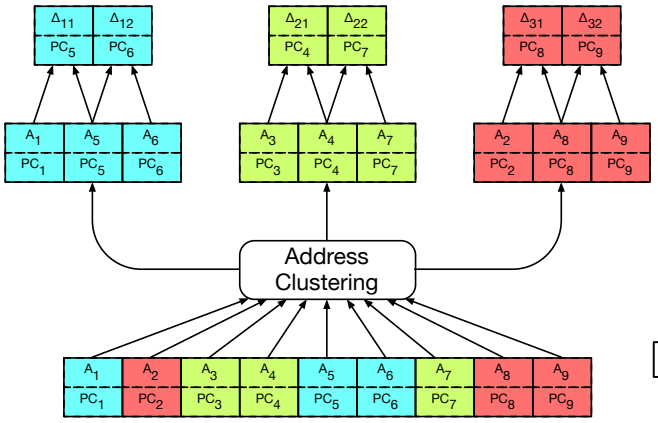
We split each trace into a training and testing set, using 70% for training and 30% for evaluation, and train each LSTM on each dataset independently. The embedding LSTM was trained with ADAM (Kingma & Ba, 2015) while the clustering LSTM was trained with Adagrad (Duchi et al., 2011). We report the specific hyperparameters used in the appendix.

5.3. Metrics

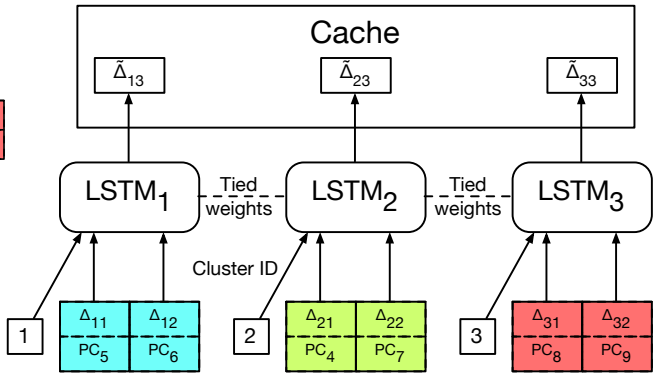
Precision We measure precision-at-10, which makes the assumption that each model is allowed to make 10 predictions at a time. The model predictions are deemed correct if the true delta is within the set of deltas given by the top-10 predictions. A label that is outside of the output vocabulary of the model is automatically deemed to be a failure.

Recall We measure recall-at-10. Each time the model makes predictions, we record this set of 10 deltas. At the end, we measure the recall as the cardinality of the set of predicted deltas over the entire set seen at test-time. This measures the ability of the prefetcher to make diverse predictions, and quantifies the percentage of the dataset that the prefetcher could predict.

One subtlety involving the clustering + LSTM model is how it is used at test-time. In practice, if an address generates a cache miss, then we identify the region of this miss, feed it as an input to the appropriate LSTM, and retrieve predictions.



(a) Clustering the address space into separate streams.



(b) The clustering + LSTM model.

Figure 4. The clustering + LSTM data processing and model.

Therefore, the bandwidth required to make a prediction is nearly identical between the two LSTM variants.

5.4. Model Comparison

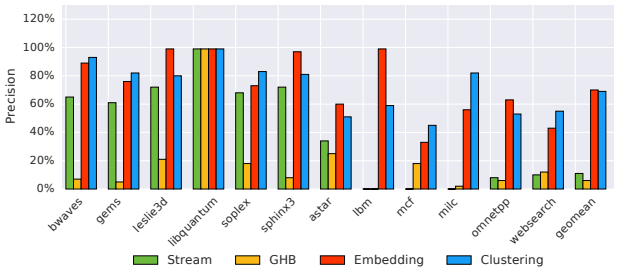
We compare our LSTM-based prefetchers to two state-of-the-art hardware prefetchers. The first is a standard stream prefetcher. We simulate a hardware structure that supports up to 10 simultaneous streams to maintain parity between the ML and traditional predictors. The second is a GHB PC/DC prefetcher (Nesbit & Smith, 2004). This is a correlation prefetcher that uses two tables. The first table stores PCs, these PCs then serve as a pointer into the second table where delta history is recorded. On every access, the GHB prefetcher jumps through the second table in order to prefetch deltas that it has recorded in the past. This prefetcher excels at more complex memory access patterns, but has much lower recall than the stream prefetcher.

Figure 5 shows the comparison of the different prefetchers across a range of benchmark datasets. While the stream prefetcher is able to achieve a high recall due to its dynamic vocabulary, the LSTM models otherwise dominate, especially in terms of precision. This is particularly apparent on the *websearch* dataset, where the complex patterns are a much better fit for the LSTM prefetchers.

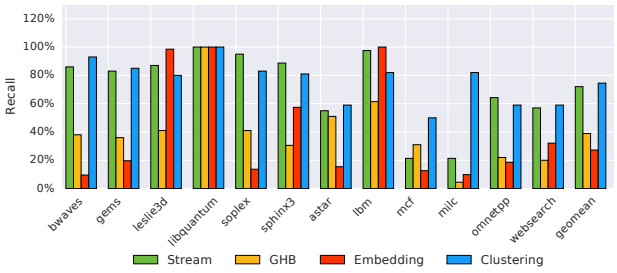
Comparing the embedding LSTM to the cluster + LSTM models, neither model obviously outperforms the other in terms of precision. The clustering + LSTM tends to generate much higher recall, likely the result of having multiple vocabularies. An obvious direction is to ensemble these models, which we leave for future work.

5.5. Predictive information of Δs vs PCs

In this experiment, we remove one of the Δs or PCs from the embedding LSTM inputs, and measure the change in predictive ability. This allows us to determine the relative information content contained in each input modality.



(a) Precision



(b) Recall

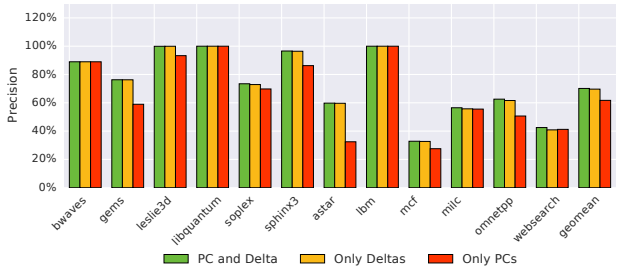
Figure 5. Precision and recall comparison between traditional and LSTM prefetchers. Geomean is the geometric mean.

As Figure 6 shows, both PCs and deltas contain a good amount of predictive information. Most of the information required for high precision is contained within the delta sequence, however the PC sequence helps improve recall.

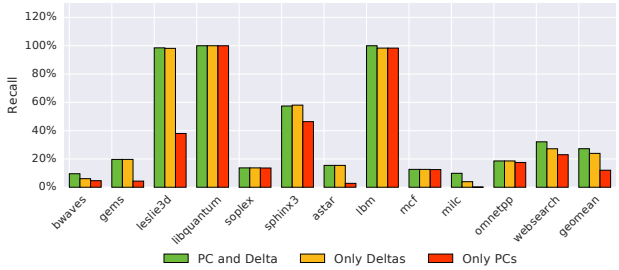
5.6. Interpreting Program Semantics

One of the key advantages of using a model to learn patterns that generalize (as opposed to lookup tables) is that the model can then be introspected in order to gain insights into the data. In Figure 7, we show a t-SNE (Maaten & Hinton, 2008) visualization of the final state of the concatenated (Δ, PC) embeddings on *mcf*, colored according to PCs.

There is clearly a lot of structure to the space. Linking PCs



(a) Precision



(b) Recall

Figure 6. Precision and Recall of the embedding LSTM with different input modalities.

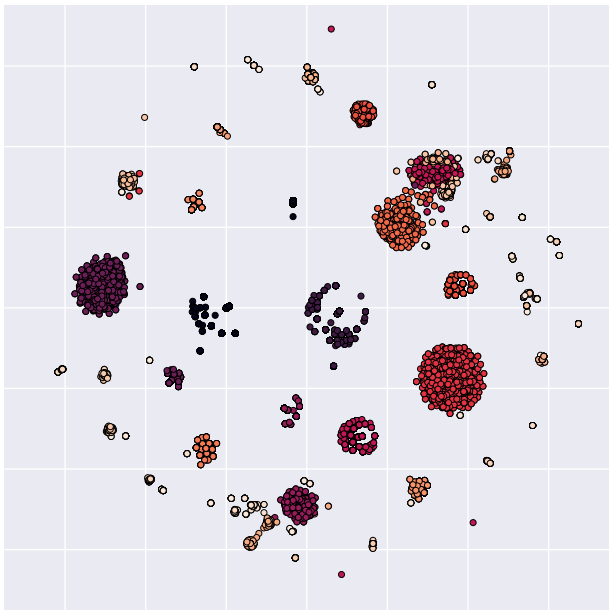


Figure 7. A t-SNE visualization of the concatenated (Δ,PC) embeddings on *mcf* colored according to PC instruction.

back to the source code in *mcf*, we observe one cluster that consists of repetitions of the same code statement, caused by the compiler unrolling a loop. A different cluster consists only of pointer dereferences, as the application traverses a linked list. Applications besides *mcf* show this learned structure as well. In *omnetpp* we find that inserting and removing into a data structure are mapped to the same cluster and data comparisons are mapped into a different cluster. We show these code examples in the appendix, and leave further inspection for future work, but the model appears to be learning about the higher level structure of the application.

6. Related Work

6.1. Machine Learning in Microarchitecture

Machine learning in microarchitecture and computer systems is not new, however the application of machine learning as a complete replacement for traditional systems, especially using deep learning, is a relatively new and largely uncharted area. Here we outline several threads of interaction between machine learning and microarchitecture research.

Prior work has also directly applied machine learning techniques to microarchitectural problems. Notably, the perceptron branch predictor (Jiménez & Lin, 2001) uses a linear classifier to predict whether a branch is taken or not-taken. The perceptron learns in an online fashion by incrementing or decrementing weights based on taken/not-taken outcome. The key benefit of the perceptron is its simplicity, eschewing more complicated training algorithms such as back-propagation to meet tight latency requirements.

Other applications of machine learning in microarchitecture include applying reinforcement learning for optimizing the long-term performance of memory controller scheduling algorithms (Ipek et al., 2008), tuning performance knobs (Blanton et al., 2015), and using bandits to identify patterns in hardware and software features that relate to a memory access (Peled et al., 2015).

Recent work also proposed an LSTM-based prefetcher and evaluated it with generated traces following regular expressions (Zeng, October 2017). Using the squared loss, the model caters to capturing regular, albeit non-stride, patterns. Irregular memory reference patterns, either due to workload behavior or multi-core processor reordering/interleaving, pose challenges to such regression-based approaches. Zeng (October 2017) evaluate their model on randomly generated patterns. As detailed in Section 3, we have found regression models to be a poor fit on real workloads.

Very recent work has also explored the usage of machine learning to replace conventional database index structures such as b-trees and bloom filters (Kraska et al., 2017). Although the nature of this problem differs from cache prefetching, there are many similarities as well. Specifically, the idea of using the distribution of the data to learn specific models as opposed to deploying generic data structures. Our clustering approach is also reminiscent of the hierarchical approach that they deploy. Importantly, they find that neural network models are faster to query than conventional data structures.

6.2. Machine Learning of Program Behavior

Memory traces can be thought of as a representation of program behavior. Specifically, they represent a bottom-up view of the dynamic interaction of a pre-specified program with a particular set of data. From a machine learning perspective, researchers have taken a top-down approach to explore whether neural networks can understand program behavior and structure.

One active area of research is program synthesis, where a full program is generated from a partial specification– usually input/output examples. Zaremba & Sutskever (2014) use an LSTM to estimate the output of a randomly generated program. The model can only see the sequence of ASCII characters representing the program, and must also generate the resulting output as a sequence of characters. Zaremba & Sutskever (2014) falls into the category of sequence-to-sequence models (Sutskever et al., 2014). Another example in this category is the *Neural Turing Machine*, which augments an LSTM with an external memory and an attention mechanism to form a differentiable analog of a Turing machine (Graves et al., 2014). This is used to solve simple problems such as sorting, copying, and associative recall.

There is also work on modeling source code directly, as opposed to modeling properties of the resulting program. For example, (Maddison & Tarlow, 2014) creates a generative model of source code using a probabilistic context-free grammar. (Hindle et al., 2012) models source code as if it were natural language using an n-gram model. This approach has been extended to neural language models for source code (White et al., 2015). Lastly, Cummins et al. use neural networks to mine online code repositories to automatically synthesize applications (Cummins et al., 2017).

7. Conclusion and Future Work

Computer architects have long exploited the benefits of learning and predicting program behavior to unlock control and data parallelism. The conventional approach of table-based predictors, however, is too costly to scale for data-intensive irregular workloads. The models described in this paper demonstrate significantly higher precision and recall than table-based approaches. This study also motivates a rich set of questions that this initial exploration does not solve, and we leave these for future research.

We have focused on a train-offline test-online model, using precision and recall as evaluation metrics. A prefetcher, through accurate prefetching, changes the distribution of cache misses, which could make a static RNN model less effective. There are several ways to alleviate this, such as adapting the RNN online, or training on both hits and misses. However, this changes the dataset and increases the computational and memory burden of the prefetcher.

There is a notion of timeliness that is also an important consideration. If the RNN prefetches a line too early, it risks evicting data from the cache that the processor hasn't used yet. If it prefetches too late, the performance impact of the request is minimal, as much of the latency cost of accessing main memory has already been paid. One simple heuristic is to predict several steps ahead, instead of just the next step. This would be similar to the behavior of stream prefetchers.

Finally, the effectiveness of an RNN prefetcher must eventually be measured in terms of its performance impact within a program. Ideally the RNN would be directly optimized for this. This and the previous issues motivate the use of reinforcement learning techniques (Sutton & Barto, 1998) as a method to train these RNNs in dynamic environments. Indeed, modern microarchitectures also employ control systems to control prefetcher aggressiveness, and this provides yet another area in which neural networks could be used.

Additionally, we have not evaluated the hardware design of our models. Correlation prefetchers are difficult to implement in hardware because of their memory size. While it is unclear if DNNs can meet the latency demands required for a hardware accelerator, neural networks also significantly compress learned representations during training, and shift the problem to a compute problem rather than a memory capacity problem. Given the recent proliferation of ML accelerators, this shift towards compute leaves us optimistic at the prospects of neural networks in this domain.

Prefetching is not the only domain where computer systems employ speculative execution. Branch prediction is the process of predicting the direction of branches that an application will take. Branch target buffers predict the address that a branch will redirect control flow to. Cache replacement algorithms predict the best line to evict from a cache when a replacement decision needs to be made. One consequence of replacing microarchitectural heuristics with learned systems is that we can introspect those systems in order to better understand their behavior. Our t-SNE experiments only scratch the surface and show an opportunity to leverage much of the recent work in understanding RNN systems (Murdoch & Szlam, 2017; Murdoch et al., 2018). Recent work has also identified timing based attacks as a vulnerability for speculative systems (Kocher et al., 2018; Lipp et al., 2018), security implications and adversarial attacks are an open problem.

The t-SNE results also indicate that an interesting view of memory access traces is that they are a reflection of program behavior. A trace representation is necessarily different from e.g., input-output pairs of functions, as in particular, traces are a representation of an entire, complex, human-written program. This view of learning dynamic behavior provides a different path towards building neural systems that learn and replicate program behavior.

References

Ayers, Grant, Ahn, Jung Ho, Kozyrakis, Christos, and Ranganathan, Parthasarathy. Memory hierarchy for web search. In *HPCA - Industrial Session*, 2018.

Bengio, Yoshua, Ducharme, Réjean, Vincent, Pascal, and Jauvin, Christian. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

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

Charney, M. J. and Reeves, A. P. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell Univ., 1995.

Cummins, Chris, Petoumenos, Pavlos, Wang, Zheng, and Leather, Hugh. Synthesizing benchmarks for predictive modeling. In *CGO*. IEEE, 2017.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12 (Jul):2121–2159, 2011.

Ferdman, Michael, Adileh, Almutaz, Kocberber, Onur, Volos, Stavros, Alisafaee, Mohammad, Jevdjic, Djordje, Kaynak, Cansu, Popescu, Adrian Daniel, Ailamaki, Anastasia, and Falsafi, Babak. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *ASPLOS*, 2012.

Gindele, J. D. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.

Golla, Robert and Jordan, Paul. T4: A highly threaded server-on-a-chip with native support for heterogeneous computing. In *Hot Chips 23*. IEEE Computer Society Press, 2011.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Gutierrez, Anthony, Dreslinski, Ronald G, Wenisch, Thomas F, Mudge, Trevor, Saidi, Ali, Emmons, Chris, and Paver, Nigel. Full-system analysis and characterization of interactive smartphone applications. In *IISWC*, 2011.

Hashemi, Milad, Marr, Debbie, Carmean, Doug, and Patt, Yale N. Efficient execution of bursty applications. *IEEE Computer Architecture Letters*, 2016.

Hindle, Abram, Barr, Earl T, Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *International Conference on Software Engineering*, pp. 837–847. IEEE, 2012.

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

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Ipek, Engin, Mutlu, Onur, Martínez, José F, and Caruana, Rich. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

Jain, Akanksha and Lin, Calvin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, 2013.

Jiménez, Daniel A. and Lin, Calvin. Dynamic branch prediction with perceptrons. In *HPCA*, pp. 197–206, 2001.

Joseph, Doug and Grunwald, Dirk. Prefetching using Markov predictors. In *ISCA*, 1997.

Jouppi, Norman. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

Kanev, Svilen, Darago, Juan Pablo, Hazelwood, Kim, Ranganathan, Parthasarathy, Moseley, Tipp, Wei, Gu-Yeon, and Brooks, David. Profiling a warehouse-scale computer. In *ISCA*, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2015.

Kocher, Paul, Genkin, Daniel, Gruss, Daniel, Haas, Werner, Hamburg, Mike, Lipp, Moritz, Mangard, Stefan, Prescher, Thomas, Schwarz, Michael, and Yarom, Yuval. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

Kozyrakis, Christos, Kansal, Aman, Sankar, Sriram, and Vaid, Kushagra. Server engineering insights for large-scale online services. *IEEE Micro*, 30(4):8–19, July 2010.

Kraska, Tim, Beutel, Alex, Chi, Ed H, Dean, Jeffrey, and Polyzotis, Neoklis. The case for learned index structures. *arXiv preprint arXiv:1712.01208*, 2017.

Lai, An-Chow, Fide, Cem, and Falsafi, Babak. Dead-block prediction and dead-block correlating prefetchers. In *ISCA*, 2001.

Lipp, Moritz, Schwarz, Michael, Gruss, Daniel, Prescher, Thomas, Haas, Werner, Mangard, Stefan, Kocher, Paul, Genkin, Daniel, Yarom, Yuval, and Hamburg, Mike. Meltdown. *ArXiv e-prints*, January 2018.

Luk, Chi-Keung, Cohn, Robert, Muth, Robert, Patil, Harish, Klauser, Artur, Lowney, Geoff, Wallace, Steven, Reddi, Vijay Janapa, and Hazelwood, Kim. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

Luong, Thang, Sutskever, Ilya, Le, Quoc V., Vinyals, Oriol, and Zaremba, Wojciech. Addressing the rare word problem in neural machine translation. In *Association for Computational Linguistics*, pp. 11–19, 2015.

Maaten, Laurens van der and Hinton, Geoffrey. Visualizing data using t-sne. *Journal of machine learning research*, 9 (Nov):2579–2605, 2008.

Maddison, Chris and Tarlow, Daniel. Structured generative models of natural source code. In *International Conference on Machine Learning*, pp. 649–657, 2014.

Mikolov, Tomas, Karafiát, Martin, Burget, Lukas, Cernockỳ, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *Interspeech*, volume 2, pp. 3, 2010.

Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, Corrado, Greg S, and Dean, Jeff. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pp. 3111–3119, 2013.

Mnih, Andriy and Hinton, Geoffrey E. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pp. 1081–1088, 2009.

Murdoch, W. James and Szlam, Arthur. Automatic rule extraction from long short term memory networks. *International Conference on Learning Representations*, 2017.

Murdoch, W. James, Liu, Peter J., and Yu, Bin. Beyond word importance: Contextual decomposition to extract interactions from LSTMs. *International Conference on Learning Representations*, 2018.

Nesbit, Kyle J. and Smith, James E. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

Oord, Aaron Van, Kalchbrenner, Nal, and Kavukcuoglu, Koray. Pixel recurrent neural networks. In *International Conference on Machine Learning*, 2016a.

Oord, Aaron van den, Dieleman, Sander, Zen, Heiga, Simonyan, Karen, Vinyals, Oriol, Graves, Alex, Kalchbrenner, Nal, Senior, Andrew, and Kavukcuoglu, Koray. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016b.

Palacharla, Subbarao and Kessler, R. E. Evaluating stream buffers as a secondary cache replacement. In *ISCA*, 1994.

PaX Team. Pax address space layout randomization ASLR. 2003. URL https://pax.grsecurity.net/.

Peled, Leeor, Mannor, Shie, Weiser, Uri, and Etsion, Yoav. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, 2015.

Roth, Amir, Moshovos, Andreas, and Sohi, Gurindar S. Dependence based prefetching for linked data structures. In *ASPLOS*, 1998.

Somogyi, Stephen, Wenisch, Thomas F., Ailamaki, Anastassia, Falsafi, Babak, and Moshovos, Andreas. Spatial memory streaming. In *ISCA*, 2006.

SPEC CPU2006. The Standard Performance Evaluation Corporation, 2006. http://www.specbench.org/.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Sutton, Richard S and Barto, Andrew G. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

White, Martin, Vendome, Christopher, Linares-Vásquez, Mario, and Poshyvanyk, Denys. Toward deep learning software repositories. In *Mining Software Repositories*, pp. 334–345. IEEE, 2015.

Wulf, Wm. and McKee, Sally. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 1995.

Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Zeng, Yuan. Long short term based memory hardware prefetcher. *International Symposium on Memory Systems*, October 2017.

Learning Memory Access Patterns

Milad Hashemi [1] **Kevin Swersky** [1] **Jamie A. Smith** [1] **Grant Ayers** [2 *] **Heiner Litz** [3 *] **Jichuan Chang** [1]
Christos Kozyrakis [2] **Parthasarathy Ranganathan** [1]

Appendix

A. Interpreting t-SNE Plots

By mapping PCs back to source code, we observe that the model has learned about program structure. We show examples from two of the most challenging *SPEC CPU2006* applications to learn, *mcf* and *omnetpp*.

[*]Equal contribution [1]Google [2]Stanford University [3]University of California, Santa Cruz. Correspondence to: Milad Hashemi <miladh@google.com>, Kevin Swersky <kswersky@google.com>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

A.1. mcf

The following function from *mcf* appears in two different t-SNE clusters:

```
1     while( node )
2     {
3         if( node->orientation == UP )
4             node->potential = node->
      basic_arc->cost + node->pred->potential
      ;
5         else /* == DOWN */
6         {
7             node->potential = node->pred->
      potential - node->basic_arc->cost;
8             checksum++;
9         }
10        tmp = node;
11        node = node->child;
12        node = tmp;
13
14        while( node->pred )
15        {
16            tmp = node->sibling;
17            if( tmp )
18            {
19                node = tmp;
20                break;
21            }
22            else
23                node = node->pred;
24        }
25    }
26
```

One cluster contains only different instances of line 4, unrolled into three different instructions at three different PCs. We show the line of code, followed by the assembly code in (PC: Instruction) format:

```
node->potential = node->basic_arc->cost
 + node->pred->potential;
401932: mov  0x18(%rdx),%rsi
401888: mov  0x18(%r10),%rsi
4018df: mov  0x18(%r11),%rsi
```

A second cluster identifies only the PCs responsible for the linked list traversal, at lines 11 and 16:

```
node = node->child;
401878: mov  0x10(%rdx),%r10
40187c: mov  %rcx,(%rdx)

tmp = node->sibling;
4019a2: mov 0x20(%r9),%rcx
```

A.2. omnetpp

We show the result of running t-SNE on the learned (Δ, PC) embeddings of *omnetpp* in Figure 1.

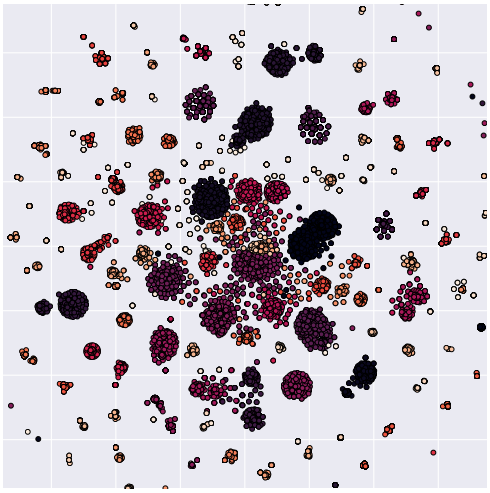


Figure 1. A t-SNE visualization of the concatenated (Δ,PC) embeddings on *omnetpp* colored according to PC instruction.

Examining some of the clusters closely, we find interesting patterns. The following code inserts and removes items into an owner's list:

```
1     // remove from owner's child list
2     if (ownerp!=NULL)
3     {
4          if (nextp!=NULL)
5               nextp->prevp = prevp;
6          if (prevp!=NULL)
7               prevp->nextp = nextp;
8          if (ownerp->firstchildp==this)
9               ownerp->firstchildp = nextp;
10         ownerp = NULL;
11    }
12    // insert into owner's child list as
      first elem
13    if (newowner!=NULL)
14    {
15         ownerp = newowner;
16         prevp = NULL;
17         nextp = ownerp->firstchildp;
18         if (nextp!=NULL)
19              nextp->prevp = this;
20         ownerp->firstchildp = this;
21    }
22
```

The main insertion and removal path are both shown in the same t-SNE cluster:

```
// Removal
nextp->prevp = prevp;
448a6a: mov  0x20(%rbx),%r12
448a6e: mov  %r12,0x20(%r10)
//Insertion
nextp = ownerp->firstchildp;
44c23a: mov  0x30(%rax),%r13
44c23e: mov  %r13,0x28(%r12)
```

omnetpp's t-SNE clusters also contain many examples of comparison code from very different source code files that are used as search statements being mapped to the same t-SNE cluster. Since these comparators are long, they get compiled to many different assembly instructions, so we only show the source code below. Lines 3 and 17 are both mapped to the same t-SNE cluster among other similar comparators:

```
1     cObject *cArray::get(int m)
2     {
3          if (m>=0 && m<=last && vect[m])
4               return vect[m];
5          else
6               return NULL;
7     }
8
9     void cMessageHeap::shiftup(int from){
10         // restores heap structure (in a
      sub-heap)
11         int i,j;
12         cMessage *temp;
13
14         i=from;
15         while ((j=2*i) <= n)
16         {
17         if (j<n && (*h[j] > *h[j+1]))   //
      direction
18              j++;
19         if (*h[i] > *h[j])  //is change
      necessary?
20         {
21              temp=h[j];
22              (h[j]=h[i])->heapindex=j;
23              (h[i]=temp)->heapindex=i;
24              i=j;
25         }
26         else
27              break;
28         }
29    }
30
```

B. Experimental Results

The experimental results for precision/recall are given in Table 1/Table 2 respectively.

C. LSTM Hyperparameters

The hyperparameters for both LSTM models are given in Table 3

D. K-Means Clustering on an Address Trace

In Figure 2 we show the results of running k-means with 6 clusters on 10^6 addresses from *omnetpp*.

Table 1. Experimental Results: Precision

Dataset	Stream	GHB	Embedding	Kmeans	Only PCs	Only Deltas
bwaves	0.65	0.07	0.89	0.93	0.89	0.89
gems	0.61	0.05	0.76	0.82	0.76	0.59
leslie3d	0.72	0.21	0.99	0.80	0.99	0.93
libquantum	0.99	0.99	0.99	0.99	0.99	0.99
soplex	0.68	0.18	0.73	0.83	0.73	0.70
sphinx3	0.72	0.08	0.97	0.81	0.96	0.86
astar	0.34	0.25	0.60	0.51	0.60	0.32
lbm	0.0001	0.0001	0.99	0.59	0.99	0.99
mcf	0.0001	0.18	0.33	0.45	0.33	0.28
milc	0.0001	0.02	0.56	0.82	0.56	0.56
omnetpp	0.08	0.06	0.63	0.53	0.62	0.51
websearch	0.1	0.12	0.43	0.55	0.41	0.41
Geometric Mean	0.11	0.06	0.70	0.69	0.70	0.61

Table 2. Experimental Results: Recall

Dataset	Stream	GHB	Embedding	Kmeans	Only PCs	Only Deltas
bwaves	0.86	0.38	0.10	0.93	0.05	0.06
gems	0.83	0.36	0.20	0.85	0.04	0.20
leslie3d	0.87	0.41	0.99	0.80	0.38	0.98
libquantum	0.99	0.99	1.00	1.00	1.00	1.00
soplex	0.95	0.41	0.14	0.83	0.14	0.14
sphinx3	0.89	0.30	0.57	0.81	0.46	0.58
astar	0.55	0.51	0.15	0.59	0.03	0.15
lbm	0.98	0.61	1.00	0.82	0.98	0.98
mcf	0.21	0.31	0.13	0.50	0.12	0.13
milc	0.21	0.05	0.10	0.82	0.001	0.04
omnetpp	0.64	0.22	0.19	0.59	0.18	0.19
websearch	0.57	0.20	0.32	0.59	0.23	0.27
Geometric Mean	0.72	0.39	0.27	0.75	0.12	0.24

Table 3. Training hyperparameters for each model.

Embedding	Network Size	128x2 LSTM
	Learning Rate	.001
	Number of Train Steps	500k
	Sequence Length	64
	Embedding Size	128
Clustering	Network Size	128x2 LSTM
	Learning Rate	.1
	Number of Train Steps	250k
	Sequence Length	64
	Number of Centroids	12

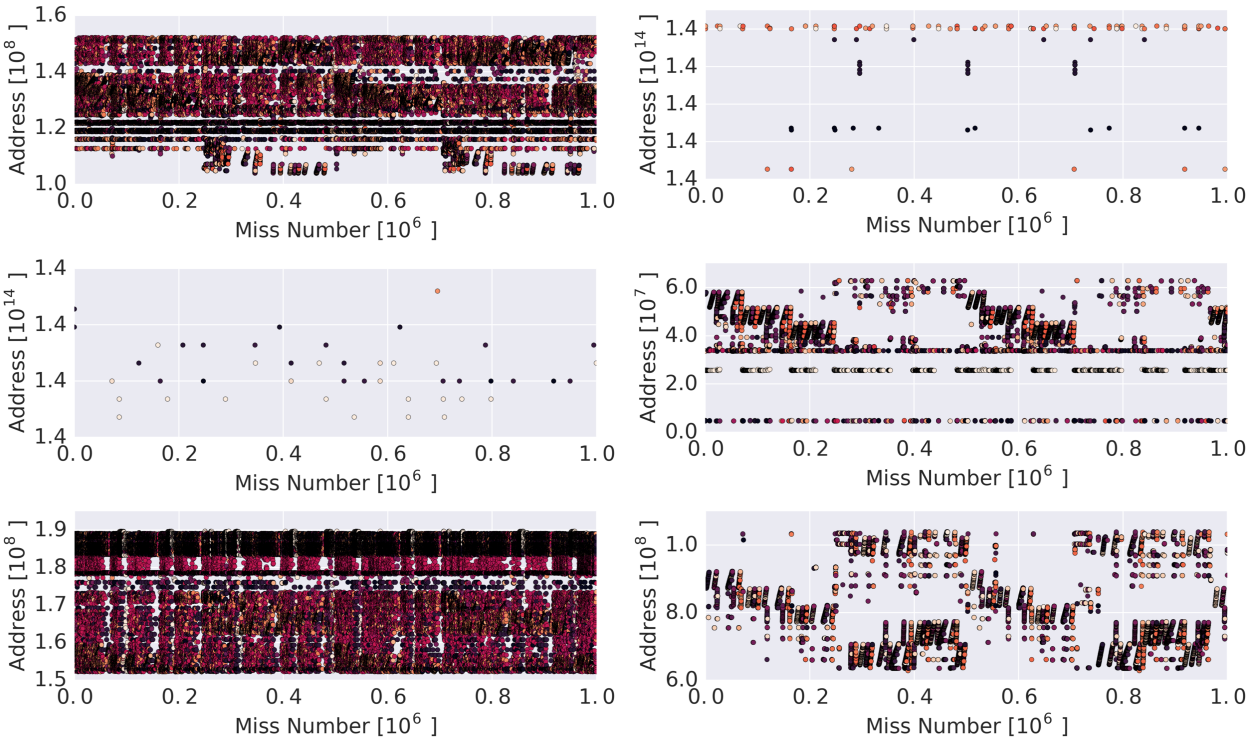


Figure 2. One million memory accesses from *omnetpp* after running k-means clustering on the address space.