Project Proposal : CS 7643
Generating Proxy memory trace using Deep Learning
Team name: Cache Overflow

Ruobing Han
Georgia Tech
Atlanta, GA, USA
rhan38@gatech.edu

Mohammad Adnaan
Georgia Tech
Atlanta, GA, USA
madnaan3@gatech.edu

Abstract

Memory trace is the sequence of memory address that is accessed by a program during execution. This is an important profiling data, which can help hardware people design hardware and provide customized optimizations for a given program. However, this data is also sensitive, which may contain the secret information and be utilized immorally by attackers. Thus, software developers cannot directly share the real memory trace with other people. Some researchers have proposed mechanism to generate proxy memory traces, which are memory trace having the similar patterns as the real memory trace, but only contain little sensitive information if any. The previous projects for generating proxy memory trace all relied on feature engineering, which required manual designing of features and the corresponding ways to capture these features. In this project, we propose two methods that use Deep Learning technology to solve this problem. Our first method is derived from seq2seq model, while the second method is based on Autoencoder. We also implement mechanism to generate dataset for training.

1. Introduction/Background/Motivation

Memory trace means the sequence of memory address accessed by a program during execution. It is an important features for program analysis. Hardware designer can design high-performance memory systems and cache systems by analyzing these traces. However, these memory traces can also be utilized by malicious attackers. With these traces, attackers can infer the algorithms used in closed-source applications [9], infer the private key used for encryption [5], rebuild the patented DNN models [4] and so on. Thus, software people can not directly share the real memory traces with hardware people, which are always the work of other departments or companies. Some researchers propose mechanisms [6, 8, 2, 1, 7] to generate proxy memory traces, which have similar pattern but do not contain any privacy information, and release these proxy traces safely.

There is a good amount of information contained in memory trace. Here, we only focus on the information useful for memory system design. For memory system optimization, the most important factor inside memory trace is the stride pattern, which is critical for designing high-performance cache and prefetcher. For example, the sequence `0,1,2,0,1,2` has the same stride pattern with sequence `23, 24, 25, 23, 24, 25`. Attackers mainly utilize the offset information in the memory sequence [9]. Thus, researchers work on capturing the stride information (relative address) for a given memory trace, and discard the offset information (absolute address) in the generated proxy trace.

2. Related work

Most related work use manually designed system to capture the stride pattern. To generate proxy memory traces for a given program, WEST [2] executed the program on simulators and recorded some critical metrics (e.g., cache miss rate, cache reuse distance), and used these metadata to generate proxy traces. This mechanism is not hardware independent, as all metrics are related with the configurations of the simulator. STM [1] directly analyzed the real memory traces to capture the stride information and generate proxy benchmarks accordingly. HALO [7] proposed another algorithm to capture stride information with higher performance. Trace Wringing [3] proposed using Hugh transformation to capture the texture information in heatmaps, which is another representation of memory trace, and used these texture information to generate proxy traces. We summarize the related works in Table 1.As we can conclude, all these related work can be regarded as feature engineering

project	down-sampling	core state	up-sampling
WEST		reuse distance set-wise temporal locality	[1] generate set index according to set locality; [2] select a block from that set according to reuse distance;
STM		reuse distance global stride table	[1] if generate a hit, use reuse distance; [2] if generate a miss, use global stride table;
HALO	split memory space to different regions	inter region reuse distance intra region stride table	[1] select a region based on inter-region reuse distance; [2] select an address in that region based on stride table;
Trace Wringing	apply houghTransformation	lines (a set of pairs of (x, y))	generate segmentations

Table 1. The summary of related work.

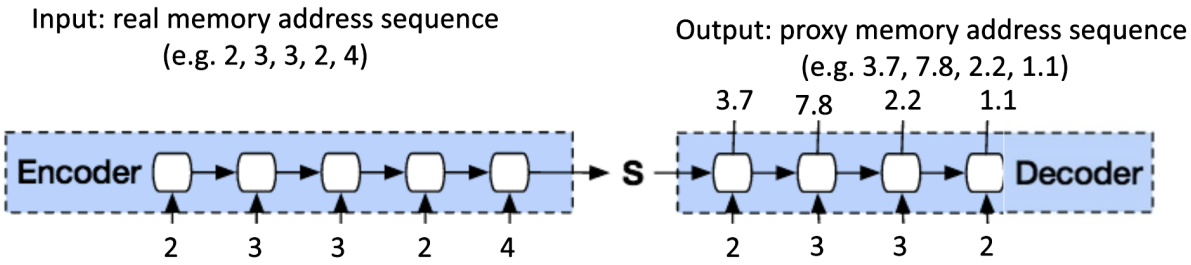

Figure 1. The overview of seq2seq model.

and generation process. The input memory trace through down-sampling, generates cores state, which only contains necessary features. After up-sampling of these features, we can generate proxy memory traces. Both Computer Vision and Natural Language Processing are traditionally feature engineering tasks, which have been rapidly developed after using Deep Learning models. Thus, we believe the tasks for generating proxy traces can also utilize Deep Learning technology.

3. Dataset

Although there is no existing dataset for memory trace, but we can easily generate them. We write a program, which access different index in an array. With different index order, we can get memory traces with different patterns. We also build a cache simulator to get the reuse distance and cache hit rate for each trace in the dataset. Codebase link for dataset generation and model implementation.

4. Approach

4.1. Seq2Seq model

In this section, we introduce our first mechanism, which uses sequence to sequence (Seq2Seq) RNN model to capture the features of real memory trace. In seq2seq architecture, the features of input real memory trace are encoded in the hidden states of the LSTM layers present inside of the encoder. The hidden states are then used by the decoder to generate proxy memory trace. The cross-entropy between generated memory trace and real memory trace was used as loss function to train the parameters of the model. The generated memory trace in this way should able to capture the trends in real memory access patterns. The overview of seq2seq model used for proxy trace generation is shown in Figure 1.

4.2. CNN model

The Seq2Seq model focuses on learning the information which is used to reproduce the real memory sequence. However, this method can not persuade users that only stride information is captured and used. Besides, this method tries to generate a proxy trace which is close to the real trace. This may also maintain privacy information in the generated proxy traces.

For hardware designer, not all stride information can be utilized. In the real hardware, only regular memory access can be captured. The most two important regular memory access are temporal locality and spatial locality. The former means programs access the same memory address within a short period, while the later means, within a short period, the programs access memory addresses which are close to each other. For seq2seq model, the DNN model will not only capture temporal/spatial locality patterns, but also other patterns. These extra patterns are not needed for proxy trace and we should discard them in proxy traces. Based on these observation, we propose another method which is based on CNN model.

4.2.1 Overview

The overview of this method is shown in Figure 2. In the first step, the input real memory trace will be converted to heatmap. Thus, in the following step, instead of handling a 1D sequence, we handle a 2D figure. The heatmap generated from the real sequence will be passed to an encoder and a decoder for down-sampling and up-sampling. After that,

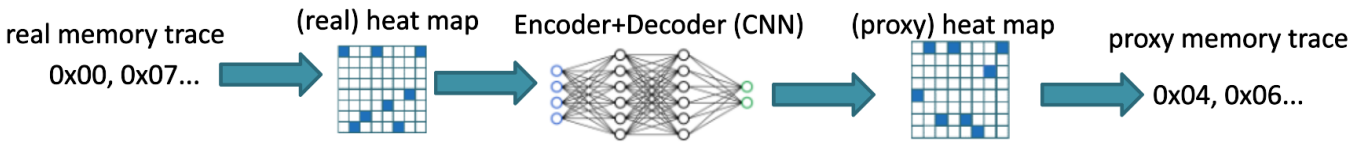

Figure 2. The overview of heatmap method.

another heatmap is generated, with the size same as the real heatmap. Finally, the proxy memory trace will be generated from the proxy heatmap. In the following sections, we describe each step in detail.

4.2.2 Heatmap

In this step, we convert the input 1D sequence into 2D heatmap. The transformation from memory sequence to heat map is shown in Figure 3. Every column in the heatmap only contains a single non-zero value. This can be regarded as a one-hot coding. In the following given example, since the first element of the memory access pattern is 0, only the first element in the first column has non-zero value. The concept of heatmap is also used in [3]. From the

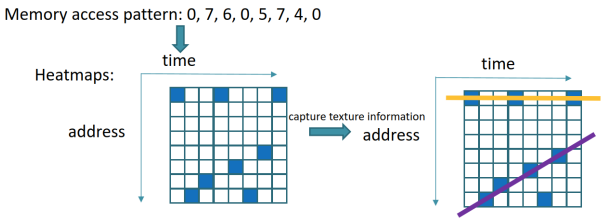

Figure 3. Sequence to heatmap.

heatmap, spatial/temporal locality can be easily captured. The yellow curve and purple curve in Figure 3 captures the temporal locality and spatial locality separately. In other word, after transforming to heatmaps, the locality information is shown as texture features. In the following steps, we rely on CNN models to capture these texture features.

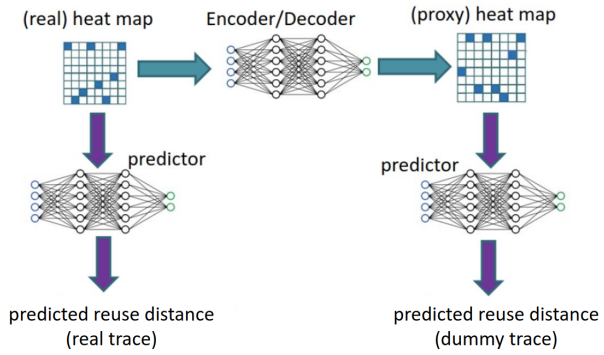

Figure 4. Predictor is used to train encoder/decoder.

4.2.3 Encoder+Decoder

The generated real heatmap will be forwarded to an encoder and a decoder. This phase is similar with auto-encoder model. The shape of the intermediate tensor is a hyperparameter. The encoder and decoder will apply down-sampling and up-sampling for the input real heatmap, and generate a proxy heatmap. Assuming the size of intermediate tensor is [batch*S] with float precision, we can guarantee that the information leaked from the generated proxy heat map is at most $\log_2 S * 32$ bits for each sample in the worst cases.

The major difference between auto-encoder and our encoder/decoder model is that we do not directly use the similarity between input (real heatmap) and output (proxy heatmap) as loss function. In our framework, we train another DNN model to predict the reuse distance for a given heatmap. And the loss function of the encoder and decoder is the similarity between input and output's predicated reuse distances. As shown in Figure 4, we pretrain another CNN as predictor, which predict the cache miss rate for a given heatmap. From the predictor, we can get the estimated cache miss rate for both real heatmap and proxy heatmap. We train the encoder and decoder to generate proxy heatmaps that have the close predicted cache miss rate as the input heatmaps.

5. Experiments and Results

The environment for evaluation is listed below:

- Software: Pytorch==1.8.0, Python==3.8.5;
- System: Ubuntu 20.0.4;
- Hardware: NVIDIA Titan XP with CUDA 10.0.

Both Seq2Seq model and CNN model are implemented by Pytorch. The seq2seq model is modified from Pytorch official example code.

5.1. Seq2Seq model

Our dataset for training consists of 10000 files of memory trace each having 128 (batch size) individual memory address. Each memory trace file have different random reuse rates and vector lengths for memory addresses. The memory trace files were generated automatically with cache

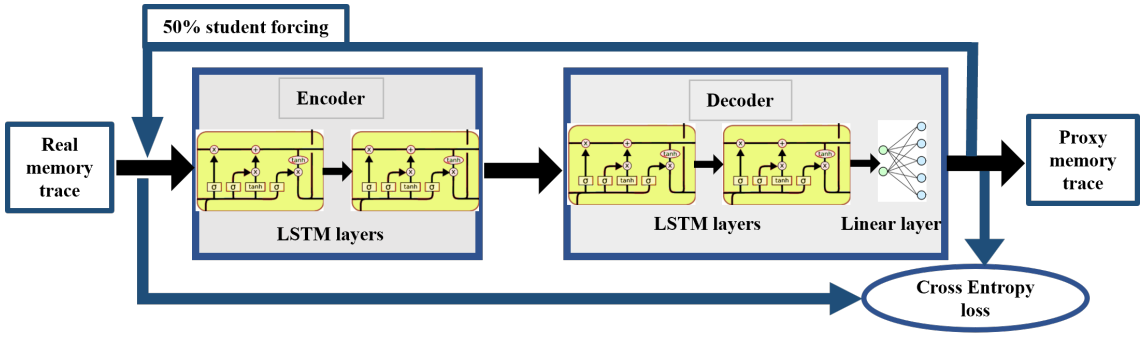

Figure 5. Structure of Seq2Seq model.

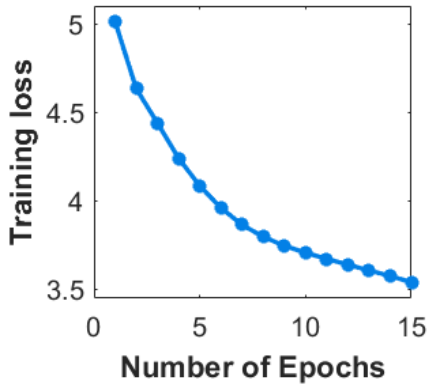

Figure 6. Training loss for Seq2Seq model.

simulator program. The structure of our seq2seq model is represented in Figure 5.

We have tested with different hyper-parameter settings for training and the best performance was obtained with 2 LSTM layers with 512 hidden dimension and 40% dropout in both encoder and decoder. The loss function output during training is shown in Figure 6.

5.2. CNN model

The CNN model contains of two DNN models. The predictor is a regression model, which predict the reuse distances for the input heatmap. The encoder down-samples the input heatmap into a tensor, and the decoder up-samples the generated tensor to generate a new heatmap. In this section, we record the evaluation of these two parts. First, we evaluate the predictor. The model configuration is listed below:

- Input: heatmap: [N, C, H, W]=[64, 1, 256, 256]
- Layer: 7xconvolution layers: kernel 3*3, following with Batch Normalization, MaxPooling, and ReLU. 2xFully connect layer in the end of the model, the hidden size are 256 and 1024.

The training curve for the predictor is shown in Figure 7.

Although we can lower the training loss and the absolute

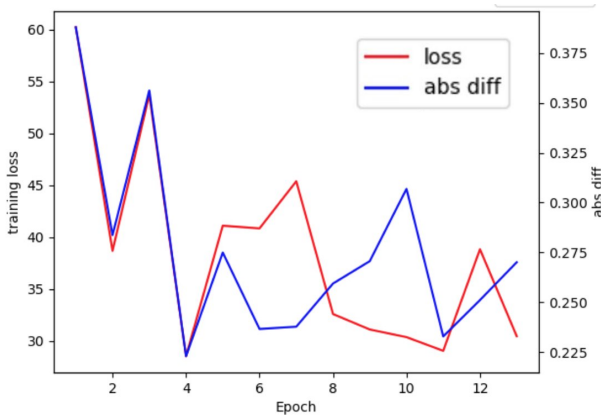

Figure 7. The training curve for the predictor.

difference, the difference is still too high to be used in industry (the SOTA results are around 0.10 in the related works). To improve the accuracy, more complex models are required. However, due to hardware limitation, we cannot train larger models.

With pre-trained predictor, we can train encoder/decoder models. The configuration of the encoder/decoder is shown below:

- Input: heatmap
- layer: encoder has 2 down-sampling blocks, each block contains a 3x3 convolution layers, following with BN, ReLU, and Maxpooling with stride equals 2; decoder has 2 up-sampling blocks, each block contains a bilinear up-sampling operation and a convolution layer, following with BN and ReLU.

The training curve is shown in Figure 8.

We also visualize the real heatmap and corresponding generated heatmaps. From the visualization result (Figure 9), we can find the encoder/decoder can learn the capture the sparse/density information for the heatmap. However, instead of density, the texture features are more important. Thus, future works are required to make the

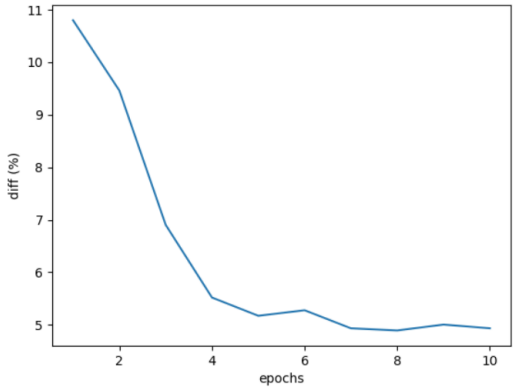

Figure 8. The training curve for the encoder/decoder.

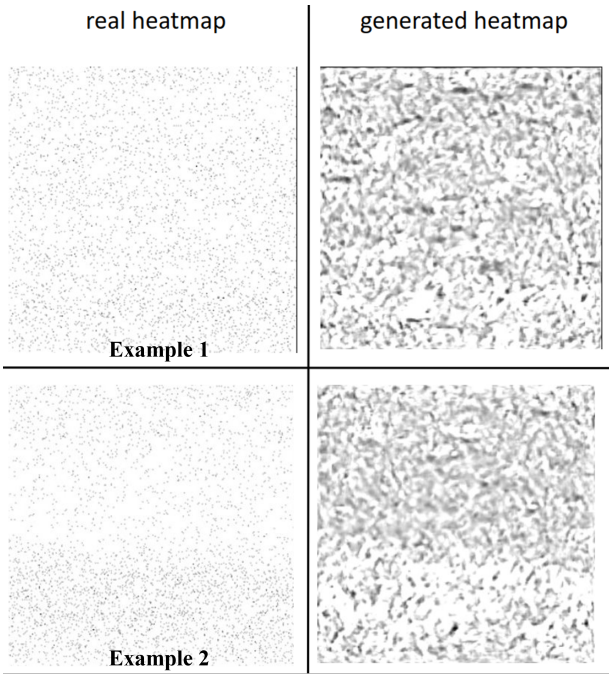

Figure 9. The visualization results for the real/generated heatmaps.

encoder/decoder more focused on texture instead of density. In a word, although during training process, the encoder/decoder can generate proxy heatmaps that achieve close predicted reuse distances, due to the low accuracy of the predictor, we cannot use the framework to generate proxy benchmark.

6. Conclusion

In this paper, we propose two mechanism to use Deep Learning to generate proxy traces. The first mechanism is based on seq2seq model. However, this model cannot persuade users that the generated proxy traces have similar pattern with the real traces and contains little privacy information, as it is a black box. Based on the observation, we propose CNN models to capture texture features, which is critical for memory system designers and do not contain sensitive information. Although we can make CNN models converge, the accuracy is still low. As the training accuracy is low, it means we need larger models. However, due to the hardware limitation, we have to remain large model training as future works.

7. Work Division

The work division is listed in Table 2.

Student Name	Contributed Aspects	Details
Ruobing Han	Generate dataset, implement CNN models	Ruobing proposes to use CNN model with memory sequence to heatmap translation. Ruobing also design the evaluation part for CNN models, and visualize the heatmap.
Mohammad Adnaan	Implementation and Analysis	Trained the LSTM of the encoder and analyzed the results. Analyzed effect of number of nodes in hidden state. Implemented Convolutional LSTM.

Table 2. Contributions of team members.

References

[1] Amro Awad and Yan Solihin. Stm: Cloning the spatial and temporal memory access behavior. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 237–247. IEEE, 2014. 1

[2] Ganesh Balakrishnan and Yan Solihin. West: Cloning data cache behavior using stochastic traces. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012. 1

[3] Deeksha Dangwal, Weilong Cui, Joseph McMahan, and Timothy Sherwood. Safer program behavior sharing through trace wringing. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1059–1072, 2019. 1, 3

[4] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. 1

[5] Tara Merin John, Syed Kamran Haider, Hamza Omar, and Marten Van Dijk. Connecting the dots: Privacy leakage via write-access patterns to the main memory. *IEEE Transactions on Dependable and Secure Computing*, 17(2):436–442, 2017. 1

[6] Ajay Joshi, Lieven Eeckhout, and Lizy John. The return of synthetic benchmarks. In *2008 SPEC Benchmark Workshop*, pages 1–11, 2008. 1

[7] Reena Panda and Lizy K John. Halo: A hierarchical memory access locality modeling technique for memory system explorations. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 118–128, 2018. 1

[8] Luk Van Ertvelde and Lieven Eeckhout. Dispersing proprietary applications as benchmarks through code mutation. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 201–210, 2008. 1

[9] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *ACM SIGOPS Operating Systems Review*, 38(5):72–84, 2004. 1