Cache Miss Rate Predictability via Neural Networks

Rishikesh Jha*
College of Informaton and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
rishikeshjha@cs.umass.edu

Arjun Karuvally*
College of Informaton and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
akaruvally@cs.umass.edu

Saket Tiwari*
College of Informaton and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
sakettiwari@cs.umass.edu

J. Eliot B. Moss
College of Informaton and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003
moss@cs.umass.edu

Abstract

Benchmark programs have been used to evaluate compilers and computer architectures (Blackburn et. al., SPEC CPU 2000 Benchmarks) which provide challenging tasks for the systems from a memory management perspective, among other things. We approach the problem of analyzing memory accesses from a machine learning perspective with the help of these benchmarks. We characterize memory accesses with a sequence of *cache miss rates* and present a new data-set for this task. The data-set draws from programs run on various JVMs, C and Fortran compilers. We work towards answering the scientific question of how predictable is a program's cache miss rate. We report the results of three distinct ANN models, which have been shown to be effective in sequence modeling.

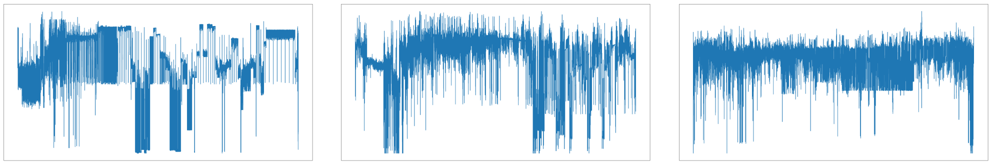
1 Introduction

Modern computers use a deep memory hierarchy to boost performance. A typical hierarchy consists of Level 1 data and instruction caches, Level 2 and 3 combined caches, and main memory. Considering virtual memory, the file system is another layer, and data and instruction translation buffers (TLBs) are further caching mechanisms. When data are not found in a particular cache, it is called a *miss*, and the *miss rate* can affect performance greatly. Here we consider the question: How *predictable* is a program's miss rate over time as it executes? This is a scientific question distinct from the engineering question of how to build a good predictor. For current hardware, it is not clear what one would do with a good predictor, except perhaps in adjusting main memory allocations in virtual memory paging.

In order to evaluate the efficiency and efficacy of compilers and computer architectures during a program run, benchmarks such as SPEC CPU 2000 [15] have been developed for C, C++, and Fortran programs, DaCapo [5] for Java programs. Traditionally, the study of benchmark programs has involved aggregate metrics and has not employed machine learning methods. Here we investigate the predictability of cache miss rates for programs from these benchmark suites.

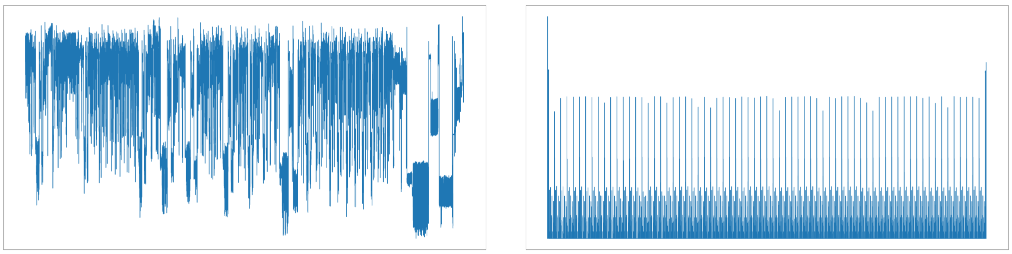
Artificial neural networks (ANNs) have been effective in learning from sequences to predict unseen patterns in Natural Language Processing (NLP) [16, 3, 13] and audio generation [17, 12]. Recently, Hashemi et al. [8] related the problem of modeling memory access pattern as a sequence learning problem in NLP.

Preprint. Work in progress.



(a) *JikesRVM* **(b)** *Jython* **(c)** *J9*

Figure 1: *JVM cache miss rates*



(a) *C* **(b)** *Fortran*

Figure 2: *JVM cache miss rates*

Chiu et al. [7] and Chiu and Moss [6] apply machine learning to *program phase change* detection and *phase prediction*. We apply ANN sequence learning techniques to study sequences of cache miss rates and determine how predictability of these sequences varies across programs. We make two contributions here. First, we introduce this sequence learning task, introducing a new data set. Second, we work towards insights into benchmark programs and compilers using sequence modeling.

Data Set: For the SPEC CPU 2000 programs ("ref" size runs) we collected traces of every memory access made by the programs in `valgrind`, specifically its Lackey tool [14]. We used the same tool on DaCapo Java programs (plus a modified `javac` benchmark) run under three Java virtual machines (JVMs): Jikes RVM [1], an IBM standard product JVM, and IBM's J9. We mapped virtual addresses to their 64-byte (or, for pages, 4096-byte) virtual cache line, and applied the least recently used (LRU) stack algorithm [4] to obtain miss rates for various cache sizes. Here we consider size 32K bytes. The rates are aggregated over windows of 1,000,000 instructions, for instruction accesses only, data only, and both. Thus we obtain six sequences of numbers in the range [0,1] from each trace (two cache line sizes $\times$ instruction only, data only, both).

Data Preprocessing: We transform the cache miss rates using $\log_{10}$ to show better the interesting miss rates close to 0. (To avoid 0 itself, we first add a small ϵ to the miss rates.) Values less than -6 we map to -6, since such small rates are insignificant. For learning, the sequences are divided into contiguous chunks, with some chunks used for training and others for testing. Training and test chunks are drawn from all parts of traces so that all program phases are captured in both training and testing.

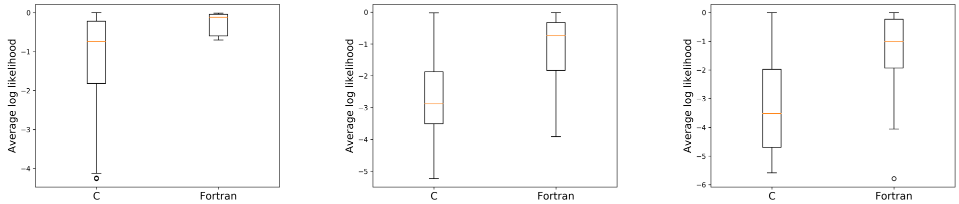
2 Models

Proposed: We model the data using three different ANNs. In keeping with the nature of the data all three models are *auto-regressive*, employ discretized representations in the input and output space and are commonly applied to sequence learning tasks across various settings.

LSTM: LSTMs [10], a variant of RNNs, have been successful in sequence modeling due to their ability to capture short and long term dependencies in sequential data [16, 18]. The model estimates the distribution

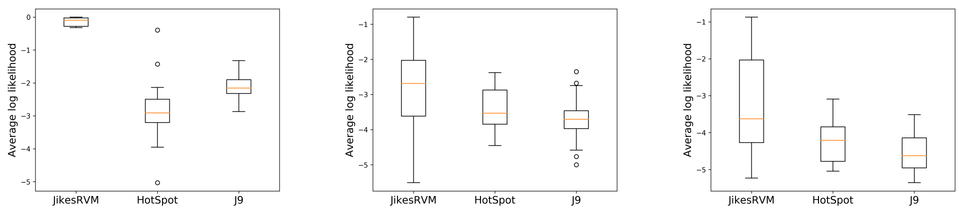
$$\Pr(\mathbf{x}) = \Pi_{t=1}^{T} \Pr(x_t | x_1, ...x_{t-1}), \tag{1}$$

where **x** is the sequence of cache miss rates, T is the length of the sequence **x**, and x_t represents the cache miss rate at time t. *Unrolling* LSTMs beyond a certain time step in the history of a sequence leads to heavy computation and *vanishing gradient* issues [9, 2], so we modified the model to account for this by unrolling only up to a finite number of time steps, h, behind the current history of the sequence. Our model consists of an LSTM layer followed by three fully connected layers [11]. The



(a) *LSTM* **(b)** *WaveNet* **(c)** *SampleRNN*

Figure 3: *C and Fortran*



(a) *LSTM* **(b)** *WaveNet* **(c)** *SampleRNN*

Figure 4: *Java virtual machines*

hidden state of the LSTM is forwarded on to the next prediction of the model. The hidden state, in theory, contains information about the values prior to the current time step.

WaveNet: Recent advances in raw audio wave form generation from text and also from preceding audio stream have been achieved using ANNs [17]. The similarity, albeit weak, between modeling raw audio wave forms and cache miss rates stems from the fact that the both sequences have high frequency components and span a finite range of values. Also, the discretization of the raw audio space has analogues in the recently proposed *neural prefetchers* [8].

We formulate the problem of predicting cache miss rates analogously to *conditional wavelets* as described by van den Oord et al. [17]. The wavelet is conditioned on **h** which is a 1-hot encoding of the memory trace ids

$$\Pr(\mathbf{x}|\mathbf{h}) = \Pi_{t=1}^{T} \Pr(x_t|x_1, ..., x_{t-1}, \mathbf{h}).$$

This one-hot encoded vector, **h**, is then projected into a dense representation which is learned at every layer:

$$\tanh(W_{f,k} * \mathbf{x} + V_{f,k}^T \mathbf{h}) \odot \sigma(W_{g,k} * \mathbf{x} + V_{g,k}^T \mathbf{h}),$$

where $V_{*,k}$ is a learnable linear projection which projects the 1-hot encoded vector into a dense representation, **x** is the output of the previous layer, $W_{*,k}$ is a matrix of learnable parameters, and the subscripts f, g represent filter and gate parameters. The scalar variable k is used to denote the layer number in the model.

We modify the WaveNet architecture, replacing the $\mu-$law encoding and decoding layers with a linear one. ($\mu-$law encoding ignores outliers, and the mid range values make fine-grained distinctions, which need not apply in our case.) We also omit stacked dilations and instead use dilations ranging from 2 to 512, increasing exponentially.

SampleRNN: Similar to WaveNet, SampleRNN [12] has achieved state-of-the-art results for audio wave form generation. In contrast to WaveNet, it uses RNNs at different time scales to model long-term dependencies instead of dilated convolutions. We use SampleRNN to model the probability distribution as described in (1). The output space is *p-way* softmax distribution.

3 Experimental Results and Insights

We discretize the sequence into 256 channels for all our three models. The models have been trained to minimize the Negative Log-Likelihood. We found the following hyperparameters worked best for our validation set:

LSTM: We use mini batches of size 7, we use a learning rate of 0.00001, we use adam optimizer with exponential decay rates β_1=0.9, β_2=0.999 and $h = 200$. **WaveNet:** We use mini batches of size 1, a learning rate of 0.001, momentum parameter 0.9, and L2 regularization with coefficient 0.001. **SampleRNN:** We use a mini batches of size 128, a *receptive field* of 512. We used the Adam Optimizer with learning rate of 0.001, β_1=0.9, β_2=0.999. Frame-sizes of 2,4,8 respectively worked best with our 3-Tier SampleRNN architecture with p = 256.

We compare the results of the three models (Figures 3 and 4). We observe Figure 4a shows how the log likelihoods of traces varies across virtual machines. Note that the 3 virtual machines have different likelihoods and the majority of the traces can be ordered as HotSpot, J9, Jikes RVM (lowest to highest likelihood). Considering log likelihoods to correspond to the predictability of the traces, Jikes RVM is seen to have the highest predictability and HotSpot the lowest. This can be attributed to the fact that Jikes RVM uses only compiled execution while HotSpot combines interpretation and compilation, so its access patterns vary more. Figure 4a in turn shows how C/Fortran traces differ in the log likelihood. Fortran programs show very high predictability compared to C programs, which are spread across the likelihood spectrum. The higher predictability of Fortran traces may be because many Fortran programs emphasize regular processing across dense arrays, while C programs lean toward pointer-linked data structures, whose accesses will be more scattered across memory.

Note that a naive predictor which predicts a uniform distribution (with probability 1/256 for each discrete bin) would result in an average log likelihood of -5.45. Although each one of our models vastly out performs such a predictor, we observe that estimating a probability distribution $\Pr(x_t|x_1, ...x_{t-1})$, for the provided data-set, is a challenging task nonetheless.

4 Discussion

We introduced a new data-set for analyzing the predictability of cache miss rates in program runs. We formulate a sequence modeling program and train three distinct models for the same. We derive inferences based on the results of these models.

For future work we can delve deeper into the scientific question and then derive more detailed inferences from the results. We need models that are engineered to predict cache miss rates and the uncertainty surrounding them. A Bayesian approach might add value in the sense they give us distribution of future values conditioned on the parameters of the model.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. A. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. F. Mergen, J. E. B. Moss, T. A. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44 (2):399–418, 2005. doi: 10.1147/sj.442.0399. URL `https://doi.org/10.1147/sj.442.0399`.

[2] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, Mar. 1994. ISSN 1045-9227. doi: 10.1109/72.279181. URL `http://dx.doi.org/10.1109/72.279181`.

[3] Y. Bengio, R. Ducharme, and P. Vincent. A neural probabilistic language model. In *NIPS*, 2000.

[4] B. T. Bennet and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4): 353–357, 1975. doi: 10.1147/rd.194.0353. URL `https://doi.org/10.1147/rd.194.0353`.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

[6] M. Chiu and E. Moss. Run-time program-specific phase prediction for Python programs. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*, pages 1:1–1:10, 2018. doi: 10.1145/3237009.3237011. URL `http://doi.acm.org/10.1145/3237009.3237011`.

[7] M.-C. Chiu, B. M. Marlin, and E. Moss. Real-time program-specific phase change detection for Java programs. In *PPPJ*, 2016.

[8] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. E. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *ICML*, 2018.

[9] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116, Apr. 1998. ISSN 0218-4885. doi: 10.1142/S0218488598000094. URL `http://dx.doi.org/10.1142/S0218488598000094`.

[10] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[11] T. Linzen, E. Dupoux, and Y. Goldberg. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535, 2016. URL `http://aclweb.org/anthology/Q16-1037`.

[12] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio. SampleRNN: An unconditional end-to-end neural audio generation model. In *ICLR*, 2017.

[13] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[14] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100, 2007. doi: 10.1145/1250734.1250746. URL `http://doi.acm.org/10.1145/1250734.1250746`.

[15] Standard Performance Evaluation Corporation. SPEC CPU 2000 benchmarks, 2000. URL `http://www.spec.org.cpu2000/`.

[16] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[17] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu. WaveNet: A generative model for raw audio. In *SSW*, 2016.

[18] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. S. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.