Memory dependency prediction with machine learning method

Kanghong Yan
kh.yan@gatech.edu
Georgia Institute of Technology

Hanning Chen
hanningchen97@gatech.edu
Georgia Institute of Technology

Jiong Feng
jfeng94@gatech.edu
Georgia Institute of Technology

Pranith Kumar
bobby.prani@gmail.com
NUVIA Inc.

ABSTRACT

The advances in core performance have outpaced the advances in memory subsystem performance over the previous decade. Utilizing Memory level parallelism (MLP) in the instruction stream plays an important role in avoiding the processor to be knee-capped by the memory subsystem. The processor speculatively issues multiple independent memory accesses to avoid stalling the pipeline. Dependent memory instructions, store-load dependencies causing RAW hazard, need to be serialized to avoid flushing the pipeline when they are speculatively executed[12]. This dependency can be dynamic based on the memory access pattern in a program. Detecting and predicting this dependency accurately helps avoid the pipeline flush overhead. Also, predicting dependency of a store-load pair is useful for implementing advanced techniques like memory renaming[9, 17], memory bypassing[13], and value prediction[11] in the load-store unit (LSU).

The current project studies a machine learning methodology to train a model based on the dependencies detected during execution and use the model to make future predictions of possible store-load dependencies. We use these predictions to implement memory disambiguation, memory renaming, and value prediction to achieve better performance by extending a store-sets[2] based dependency predictor.

CCS CONCEPTS

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

, ,

https://doi.org/10.1145/1122445.1122456

KEYWORDS

machine learning, neural networks, computer architecture, memory dependency, address disambiguation

ACM Reference Format:
Kanghong Yan, Hanning Chen, Jiong Feng, and Pranith Kumar. 2021. Memory dependency prediction with machine learning method. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The advances in core performance have outpaced the advances in memory subsystem performance over the previous decade. Utilizing Memory level parallelism (MLP) in the instruction stream plays an important role in avoiding the processor to be knee-capped by the memory subsystem. The processor speculatively issues multiple independent memory accesses to avoid stalling the pipeline. Dependent memory instructions, store-load dependencies causing RAW hazard, need to be serialized to avoid flushing the pipeline when they are speculatively executed [12]. This dependency can be dynamic based on the memory access pattern in a program. Detecting and predicting this dependency accurately helps avoid the pipeline flush overhead. Also, predicting dependency of a store-load pair is useful for implementing advanced techniques like memory renaming [9, 17], memory bypassing [13], and value prediction [11] in the load-store unit (LSU).

The current project studies a machine learning methodology to train a model based on the dependencies detected during execution and use the model to make future predictions of possible store-load dependencies. We use these predictions to implement memory disambiguation, memory renaming, and value prediction to achieve better performance by extending a store-sets[2] based dependency predictor.

2 BACKGROUND AND RELATED WORK

Memory Dependency

Memory dependency refers to alias between memory instructions and has a characteristic of non-statically determinable.

Such non-static dependency arises with memory instruction because target address is indirectly deduced from a register operand and immediate value rather than directly specified in the instruction encoding. For example, considering a following scenario that a later load might depend on an early store as long as they are referring to same target address.

$$1: \text{STORE } \$1, 2(\$2) \quad \#Mem[R2+2] <= R1$$
$$2: \text{LOAD } \$3, 4(\$4) \quad \#R3 <= Mem[R4+4]$$

Figure 1: Possible dependency in memory instructions

Here, the store instruction writes a value to the memory location specified by the value in the address ($R2 + 2$), and the load instruction reads the value at the memory location specified by the value in address ($R4+4$). The dependency between these two instruction cannot be statically determined prior to execution because the address depend on the values in $R2$ and $R4$. If the addresses are different, the instructions are independent and can be successfully executed out of order. However, if the addresses are the same, then the load instruction is dependent on the store to produce its value. This is also known as an ambiguous dependence.

Dependency Prediction

To improve performance, modern out-of-order processors typically employ speculative execution of load-store dependency: if a later load has address calculated, it would compare with early stores to find any dependency. If the address of latest store is not clear, then a prediction of dependency is given. When prediction is negative, meaning that there is no dependency, then the processor will speculatively send the load to cache, otherwise the load is marked to wait. Previous research [2] has shown that speculative execution of load leads to significant performance. Hence, accurate prediction of this dependency is critical to processor's performance as it improves memory level parallelism (MLP).

Several mechanisms have been proposed to deal with the dependency prediction. One simple way to get prediction is to memorize all load instructions that have been found to be dependent on a prior store. For example, the Alpha 21264 [10] proposes a bookkeeping structure called $WaitTable$ to record this information. However, this method doesn't consider a scenario when a load's dependency changes overtime as the program proceeds. Some researches give more accurate prediction by using information of dependent load-store pairs. Chrysos and Emer comes up with $StoreSets$ [2] which groups loads and stores that have dependency in the past. Their method correlates multiple loads and stores to give prediction. If a load and a prior store is in the same store set, then the processor will enforcing in-order issue of loads and stores to avoid possible memory violations.

Machine Learning in Computer Architecture

Machine learning has been very popular over these years. Some of its ideas has been employed in computer architecture field. For instance, perceptron-based branch predictor was in [7] and evolved [6, 8, 16] in recent years. Some companies even employ perceptron-based branch predictor in their products [3]. Besides, machine learning is also active in fields such as prefetching [14] and DRAM scheduling [5].

However, as far as we know, machine learning method has not been applied to memory dependency prediction problem yet and we would like to explore its possibilities. In this paper, we will first develop a complex sequence to sequence offline neural network to do prediction. After that, we distill insights from offline model by analysis and visualization. Finally, we propose a novel machine-learning based prediction model to give predictions on memory dependency.

3 OFFLINE SEQ2SEQ MODEL

In this section, we explain our high-level offline encoder-decoder model targeting memory dependency prediction problem.

Recurrent Neural Networks

Recurrent neural network (RNN) is quite popular in deep learning area recently, especially for Natural Language Processing (NLP) and speech recognition applications. It is designed to deal with sequential data that have correlation in time series. For example, RNN can be used to performance sequence transformation, ranging from one-to-one, one-to-many, to even many-to-many problem.

Generally, there are three variants of RNN unit: vanilla RNN, Long-Short-Term Memory (LSTM) [4] and Gated Recurrent Units (GRU) [1]. Among these three variants, vanilla RNN has simplest architecture but is bordered by vanishing and exploding gradient problems that lead to worse performance when dealing with long-term memory dependency data. LSTM, however, solves this issue with a direct feedback channel called *Cell State* to bypass the weights multiplication and thus, could remember long-term dependency data. From this aspect, LSTM often outperforms vanilla RNN in many applications. Meanwhile, GRU also has its method to remember long-term memory data and is regraded to have similar performance as LSTM [15]. Compared to LSTM, the advantages of GRU lie in its simpler architecture, fewer parameters and less computational operations.

The high-level idea of GRU is to have two gates for controlling the past information flow: one is *Update Gate* and the other is *Reset Gate*. Update gate decides how much of the past information should be kept while reset gate is used to decide

how much of the past information to forget. Fig.2 shows the

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$n_t = \phi_n(W_n x_t + U_n(r_t \odot h_{t-1}) + b_n)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot n_t$$

Figure 2: GRU equations

underlying mechanism of GRU. Variables x_t and h_t refer to input and hidden state at time step t, respectively. Variables z_t represents *Update Gate* vector and r_t corresponds to *Reset Gate* vector. Parameters W, U and bias b are learned during training by back-propagation. σ means sigmoid function and ϕ is an activation, usually hyperbolic tangent or ReLU.

Because of the performance as well as cost of GRU, we choose GRU to be underlying RNN unit in our sequence to sequence model.

Prediction Problem Abstraction

Because the PCs of load and store instruction are correlated in time and present in sequential order, memory dependency prediction can be viewed as a kind of sequence to sequence transformation, giving its input-output relationship. The model takes a stream of PCs within an instruction window (IW) with specific size as inputs and output a sequence of binary decisions to indicate whether dependency exist or not, as shown in Fig.3.

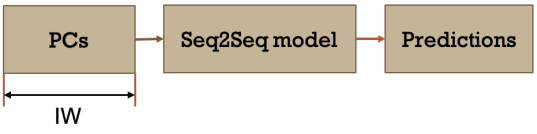

Figure 3: Overall structure of prediction model

All predictions are made with respect to the last load instruction of the input instructions stream. In other words, once the model encounter a new load instruction, it will take all memory access instructions within that instruction window as input and give predictions. For the ease of training and validation, we want to keep a constant size of input. But at the same time, we need the size of input vector large enough to handle an extreme case where all instructions in the instruction window are memory access instructions. Hence, the input vector size is set to instruction window size.

Fig.4 gives an example to illustrate the model's input and output abstraction. The left-hand side of the Fig.4 shows the current instruction window when encountering a new load with PC equals to 11. The right-hand side represents the input and output vectors of the model. The size of instruction window in this example is set to be 11.

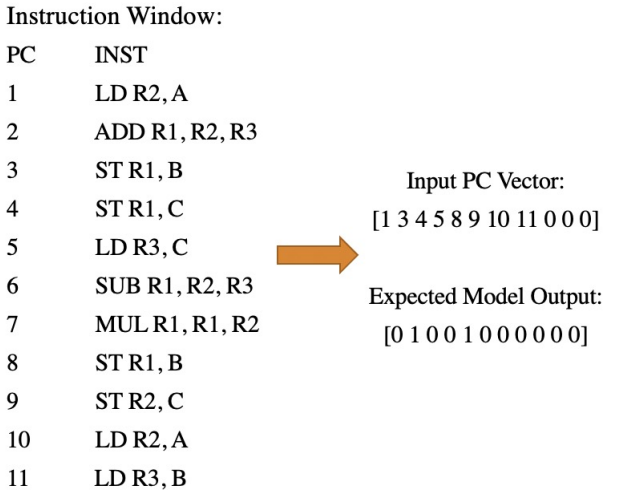

Figure 4: Example of model's input and output abstraction. The left-hand side of arrow shows current instruction window and the right-hand side shows abstract input and output. Output predictions are made with respect to the last load of instruction window

Let's illustrate the abstraction of input vector first. As mentioned before, the size of input vector will be equal to the size of instruction window. Besides, only the PC of memory instructions are kept and the other types of instruction are filtered out. Hence, the input PC vector shown in Fig.4 only contains PCs of load/store instructions, which are $\{1, 3, 4, 5, 8, 9, 10, 11\}$. Since some instructions are filtered out, the rest of input PC vector are filled with zeros. For output predictions, it has the same size as input vector and one-to-one positional relationship with input vector. If the i^{th} bit of output predictions is set to one, then it means that the i^{th} PC in the input vector has memory dependency with the last load in current instruction window. Besides, as we mentioned previously, the output is made with respect to the last laod, which is PC 11 in Fig.4. As we can see in the example, the load instruction (PC=11) is reading from memory address B. Hence, all the stores in that instruction window who also work on memory address B will be marked with dependency exists. In other words, the second and fifth bits of output vector are set to one because the instruction 3 and 8 are store instructions working on memory address B, as shown in Fig.4.

Seq2Seq model with GRU

Since the memory dependency prediction problem can be viewed as a sequence to sequence transformation, we use an encoder-decoder architecture to solve this problem. The

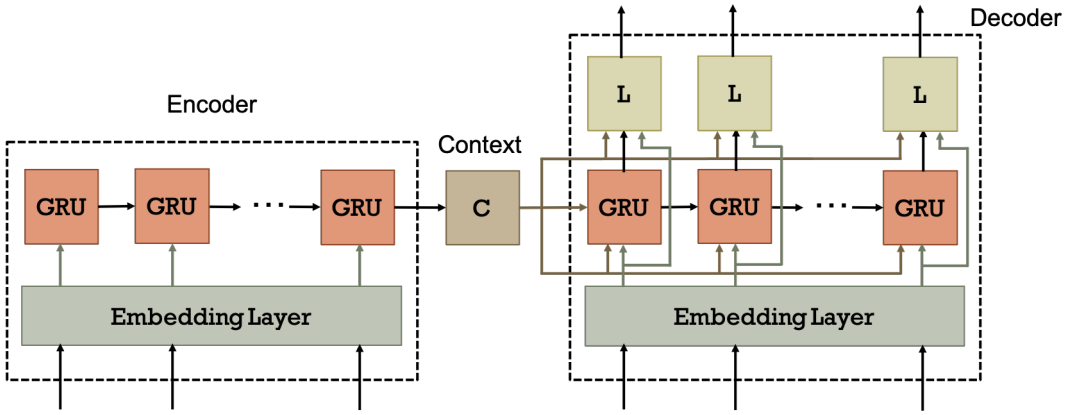

Figure 5: Seq2Seq model with GRU

encoder-decoder architecture is proposed by Kyunghyun in [1] and has been widely adopted, even in google translate service [18].

Fig.5 illustrates the block design of our Seq2Seq model. The model mainly is constituted by three parts: encoder, context vector and decoder. Both encoder and decoder use embedding layer to compress or distill from the input feature space. In this model, encoder uses an ordinary design, just concatenating series of GRUs. The output of encoder is the context vector. It represents the correlated information of input sequence. Context vector is essential because it acts like a high-level buffer with abstract global history information distilled from current instruction window. Decoder consumes context vector as a representation of source information. Context vector is reused for every time-step in the decoder for both GRU and linear layer inputs. Hence, the GRU's hidden state does not need to remember information about the source sequence but only analysis what prediction it has generated so far. Besides, the output of embedding layer is also routed to linear layer so that linear layer directly sees what the previous prediction is, without having to get this information from the hidden state. By this mean, we alleviate the information compression needed in common encoder-decoder design.

Training Result

We implement seq2seq model on Georgia Tech PACE-ICE system. The trace that we choose, are based on ChampionSim from TAMU, which is a subset of Spec2017. For each trace, we divide it into two parts: one for training and the other for validation.

The general training parameter is shown in Fig. 6. The shuffle is turned off because for dependency prediction, sequential relationship is quit important and we don't want to miss that factor. As the result, Fig.7 shows the training loss of

```
1
2  # general setting for both train and validation
3  General:
4    epochs: 5
5    shuffle: False # shuffle data or not
6    # model config
7    model: "SimSeq"
8    save_best: True # save the best model during training
9    # loss config
10   loss_type: CE # NLL, CE or Focal
11   reweight: False  # True with Focal Loss
```

Figure 6: General training parameters

five selected traces under default training parameters. And the training perplexity result is shown in Fig.8. As we can see, the training loss is quite low and the preplexity is closed to one. This implies the high accuracy of our prediction model.

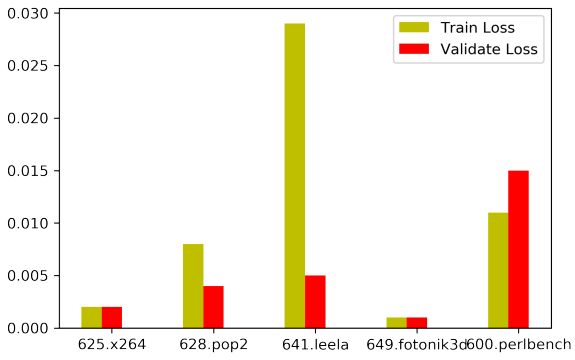

Figure 7: The training loss using default parameter

Insights from Offline Model

To maximize the performance of the model, we have a deep study of those hyperparameters including: batch size, learning rate, regulator, clop threshold etc.

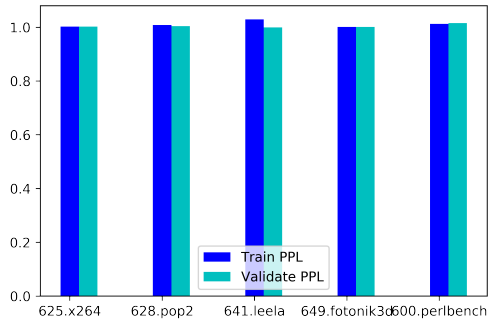

Figure 8: The training perplexity using default parameter

Because of the timing limitation, we don't have enough time to explore weights analysis and other visualization methods to understand why it works well.

4 HARDWARE IMPLEMENTATION

Due to time restrict, here we present some key ideas of hardware design.

Design Ideas

Considering the number of parameters and complexity of the architecture, it is almost impossible to directly implement seq2seq model in hardware efficiently. Hence, we want to gain essential insights from Seq2Seq model instead.

After analyze the offline training model, we find three key ideas:

- The first is that one implication is to have both global and local information. Global information represents program memory access behavior pattern and Local information indicates correlation between two instructions
- Second, we need to keep important global information including past accessed memory address and past load and store PC.
- The third is that we can use a perceptron-like design, a weight table, to indicate the confidence of whether a load and store has dependency.

5 EVALUATION

Dataset analysis

This chapter will illustrate the analysis of Dataset. In details, we would like to introduce the relationship between dependency and IW size.

After finishing the traverse of instruction window, the program will generate a matrix which is made of single vectors of each load/store instruction. As described before, the vector will illustrate the dependency that this instruction has. Thus, the generating matrix will reflect the total dependency of the corresponding instruction set. And also, most of the elements of the matrix have 0 value, which means that it is the sparse matrix that this program could generate.

As we know, the size of instruction window will depend on the size of load/store buffer. Thus we want to find the relationship between IW size and the sparsity of the matrix. We mainly test 4 benchmarks and arrange IW size vary from 10-300. The results will be illustrate as followed:

As we can see, the sparsity will decrease as we expand the size of instruction window. It may be because that the load/store instruction will have rare dependency with the more previous instruction.

6 FUTURE WORK

Due to the limited time, we only present the complex offline model and some hardware design insights in this work. There are still some important future explorations yet to be done.

- The detailed hardware design is yet to be determined
- Evaluate HW implementation on MacSim or Gem5
- Compared performance with store-set implementation

7 CONCLUSION

Memory dependency problem is well-known and has been recognized as performance critical issue. Given the magic power of machine learning, we apply a NLP method on memory dependency problem. Firstly. memory dependency problem is regarded as a kind of sequence to sequence transformation and we use encoder-decoder architecture with GRU to make predictions. Datasets are generated from Spec2017 and the Seq2Seq model exhibits high accuracy in memory dependency predictions for those datasets. After that, we analyze the model and propose essential hardware design ideas from this complex offline Seq2Seq model. Besides, we also do analysis on datasets themselves to better understand the data and use reasonable hyperparameters.

REFERENCES

[1] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[2] George Z. Chrysos and Joel S. Emer. 1998. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain) *(ISCA '98)*. IEEE Computer Society, USA, 142–153. https://doi.org/10.1145/279358.279378

[3] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya. 2020. Evolution of the Samsung Exynos CPU Microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 40–51. https://doi.org/10.1109/ISCA45697.2020.00015

[4] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

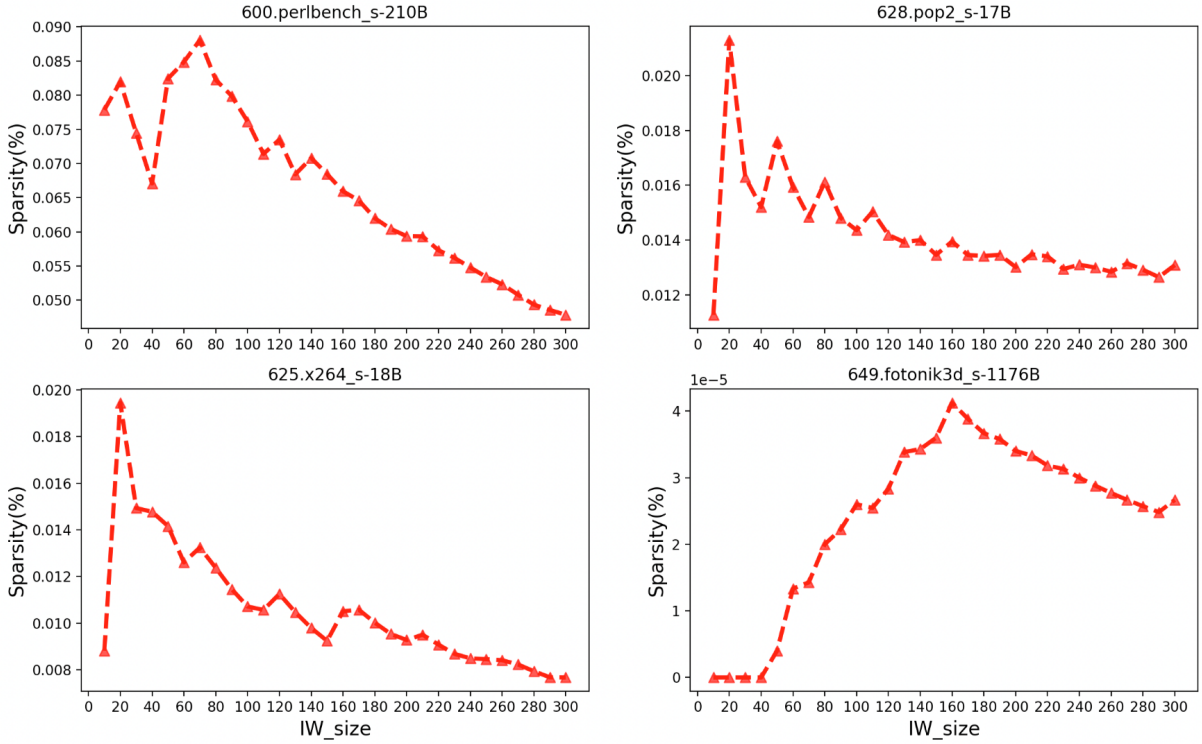

Figure 9: Dataset Analysis from Spec2017

[5] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *2008 International Symposium on Computer Architecture*. 39–50. https://doi.org/10.1109/ISCA.2008.21

[6] D. A. Jimenez. 2003. Fast path-based neural branch prediction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 243–252. https://doi.org/10.1109/MICRO.2003.1253199

[7] D. A. Jimenez and C. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 197–206. https://doi.org/10.1109/HPCA.2001.903263

[8] D. A. Jiménez. 2011. An optimized scaled neural branch predictor. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*. 113–118. https://doi.org/10.1109/ICCD.2011.6081385

[9] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. 1998. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. 216–225. https://doi.org/10.1109/MICRO.1998.742783

[10] R. E. Kessler, E. J. McLellan, and D. A. Webb. 1998. The Alpha 21264 microprocessor architecture. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*. 90–95. https://doi.org/10.1109/ICCD.1998.727028

[11] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 138–147. https://doi.org/10.1145/237090.237173

[12] Andreas Moshovos and Gurindar S. Sohi. 1997. Streamlining Inter-Operation Memory Communication via Data Dependence Prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Research Triangle Park, North Carolina, USA) *(MICRO 30)*. IEEE Computer Society, USA, 235–245.

[13] Andreas Moshovos and Gurindar S. Sohi. 1999. Speculative Memory Cloaking and Bypassing. *Int. J. Parallel Program.* 27, 6 (Dec. 1999), 427–456. https://doi.org/10.1023/A:1018776132598

[14] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 285–297. https://doi.org/10.1145/2749469.2749473

[15] Mirco Ravanelli, Philemon Brakel, Maurizio Omologo, and Yoshua Bengio. 2018. Light Gated Recurrent Units for Speech Recognition. *IEEE Transactions on Emerging Topics in Computational Intelligence* 2, 2 (2018), 92–102. https://doi.org/10.1109/TETCI.2017.2762739

[16] David Tarjan and Kevin Skadron. 2005. Merging Path and Gshare Indexing in Perceptron Branch Prediction. 2, 3 (Sept. 2005), 280–300. https://doi.org/10.1145/1089008.1089011

[17] Gary S. Tyson and Todd M. Austin. 1997. Improving the Accuracy and Performance of Memory Communication through Renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Research Triangle Park, North Carolina, USA) *(MICRO 30)*. IEEE Computer Society, USA, 218–227.

[18] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto

Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv:cs.CL/1609.08144