The Case for Learning Machine Language

Guangda Liu
Shanghai Jiaotong University
gd.liu@sjtu.edu.cn

Chieh-Jan Mike Liang
Microsoft Research
liang.mike@microsoft.com

Shijie Cao
Microsoft Research
shijiecao@microsoft.com

Shuai Lu
Microsoft Research
shuailu@microsoft.com

Leendert van Doorn*
Qualcomm
leendert@paramecium.org

Abstract

This paper focuses on enabling modern processors to better predict upcoming instructions that will be executed, in order to improve instruction-related speculations at runtime. Using branch prediction as a case study, we take the first step to motivate the potential of learning semantic correlations in machine language (i.e., CPU instructions), and we demonstrate how to apply language modeling to machine language. Although various approaches have been proposed for instruction-related runtime speculations, they remain general-purpose and rely on language-agnostic features. Furthermore, we present a branch predictor design that takes advantage of our Transformer-based language model. Empirical results from SPEC-CPU-2017 benchmarks (on RISC-V) show that language modeling can improve the branch prediction accuracy by up to 11.03%, and the processor IPC by up to 21.16%.

1 Introduction

Performance of modern processors depends on a spectrum of instruction-related runtime speculations, to minimize stalls waiting for instructions to be loaded for execution. These speculations cover branch prediction, instruction cache replacement and pre-fetching, branch target buffer (BTB) pre-fetching, and so on. Over the years, various hand-crafted heuristics [20, 8, 11, 19, 15, 7] and machine learning techniques [14, 12, 25, 21, 17] have been proposed and deployed to separately drive these speculations. The common practice is to leverage temporal locality in instruction executions, to infer upcoming instructions. Taking branch prediction as an example, the history of whether a branch was previously taken can be an indicator of how likely it will be taken in the future.

Yet, existing efforts to drive instruction-related speculations remain general-purpose, and they largely rely on language-agnostic features (e.g., the observed runtime history mentioned above). In contrast, we posit the key is to take advantage of the program-specific semantic correlations in machine language. Here, we use the term, *machine language*, to refer to processor-targeted instructions that compilers translate from high-level programming languages. Intuitively, the occurrence of an instruction should exhibit a high correlation with its preceding instructions, especially that the sequence of processor instructions should preserve some patterns in the user-written program.

To this end, this paper explores the extent to which learning semantic correlations in the machine language can improve processor performance. We present LM-ML (Language Model for Machine Language), which takes the first step to apply language modeling to predict upcoming instructions that processor will execute. Our problem formulation is analogous to the high-level code completion task [6, 18, 5, 10]. Given a sequence of instructions, LM-ML uses transformers to predict the next I

*This work was done when Guangda Liu and Leendert van Doorn were at Microsoft.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

instructions. While our approach can be applied to instruction-related speculations in general, this paper takes branch prediction to demonstrate the potential of LM-ML. On SPEC-CPU-2017 [23] benchmarks, empirical results show that LM-ML can improve the branch prediction accuracy by up to 11.03% and 8.87%, as compared to the industry-deployed Bimodal [22] and Perceptron [14] baselines, respectively. In turn, LM-ML improves the processor performance, or IPC (instructions per cycle), by up to 21.16% and 17.66%, as compared to the two baselines above.

2 Background and Motivation

2.1 Execution Stalls in Modern Processors

Execution stalls are inherent, as modern micro-architecture mainly follows the Von Neumann architecture that decouples the computation unit (i.e., processor) and memory unit. There are two sources of stalls. First, processors need to load instructions (i.e., processor-targeted opcodes) from memory, prior to the execution. Second, modern processors execute instructions in a pipeline of multiple stages (e.g., fetch, decode, execute, and write-back stages). If the subsequent instruction were to be fed into the execution pipeline only upon the completion of the previous instruction, the pipeline would have idle stages, hence stalls. Stalls lower the processor performance, which is typically quantified by the number of executed instructions per cycle (IPC).

Branch prediction. To mitigate stalls, modern processors rely on a spectrum of instruction-related runtime speculations. One prominent example is branch prediction — it tries to fill up the processor's execution pipeline without waiting for branching instructions to be fully resolved. The challenge lies in speculating whether an upcoming (but not yet executed) branch would be taken or not. Branch mis-predictions imply that processor would pre-fetch wrong instructions. This results in stalls, as the pipeline then needs to be flushed and re-loaded with correct instructions.

In the rest of this paper, we use branch prediction as the case study. Branch mis-predictions are a major factor of processor inefficiency. Lin et al. [16] measured that mis-predictions lower the processor IPC by 18%, on SPEC-CPU-2017 benchmarks [23]. Since the number of instructions flushed is proportional to the pipeline length, the problem exacerbates as modern processor design tends to have a long pipeline [9].

2.2 Related Work

Existing approaches for instruction-related speculations remain general-purpose, and they largely rely on language-agnostic features. In the case of branch prediction, most approaches rely on observing the per-branch history at runtime. For example, the single-bit-counter approach keeps the last branching outcome of a branch, and Bimodal expands the state space to keep a branch's last two outcomes [22]. TAGE [20] introduces multiple lengths of per-branch histories, to better fit different branches. Finally, recent efforts utilize neural networks to learn patterns in the past branching outcome — Perceptron [14] trains predictors to assign a weight to each positions in the history, and calculate the inner product to predict the branching direction.

Unfortunately, approaches relying on language-agnostic features can exhibit a diminishing return, where straightforward scaling of the resources to record longer branching history might only marginally improve the prediction accuracy. Specifically, Lin et al. [16] observed that increasing TAGE storage by a factor of 8 results in only 2.7% gain in IPC. Zouzias et al. [26] observed that increasing Perceptron's history length beyond 60 offers only negligible accuracy improvement.

3 Semantic Correlations in Machine Language

Rather than proposing yet-another language-agnostic approach, this paper explores how learning semantic correlations in machine language can better drive instruction-related speculations. We use the term *machine language*, to refer to processor-targeted instructions that compilers translate from high-level programming languages. Instructions encode processor-targeted opcodes (e.g., RISC-V [2]), which are the features we learn.

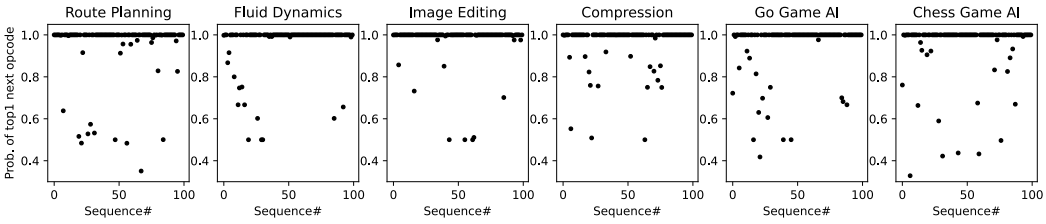

Figure 1: Machine language exhibits strong semantic correlations, i.e., how likely a sequence of executed instructions can indicate the next instruction. These figures show the top-1 opcode probability, for the 100 most frequent sequences in each benchmark. An average of 85.83 and 98.83 sequences have a top-1 opcode probability of 100% and >50%, respectively.

3.1 Characterizations of Machine Language

We define this semantic correlation by P_{next}, which is the probability of the most frequent instruction opcode, given S (a sequence of s previously executed instructions). P_{next} can be formulated as the top-1 opcode probability, as follows.

$$P_{next}(S) = \underset{c \in CPU\_Opcodes}{max} P(I_{s+1} = c \mid S = (I_1, I_2, ...I_s)) \quad (1)$$

If the semantic correlation is high in machine language, the implication is that it becomes feasible to predict upcoming instructions (i.e., I_{s+1} above) based on previously executed instructions (i.e., S above). To this end, this section first characterizes semantic correlations in real-world applications, and then discusses how we can leverage recent advancements in NLP.

We analyze traces of instruction opcodes that processors execute, for six benchmarks in SPEC-CPU-2017 [23]. These benchmarks are based on real-world applications, which include File Compression, Image Editing, Route Planning, 3D Rendering, Chess Game AI, and Go Game AI. From execution traces, we quantify their semantic correlations (P_{next} above), for sequences of s=8 instructions.

Fig 1 illustrates top-1 opcode probability, for the 100 most frequent instruction sequences in each benchmark. These sequences appear >155,000 times in our traces. We make the following observations. First, an average of 85.83 sequences have a top-1 opcode probability of 100%. In other words, given these sequences, we can be certain of their subsequent instruction. Second, an average of 98.83 sequences have a top-1 opcode probability over 50%. Since this probability is over 50%, the uncertainty is low enough for machine learning algorithms to accurately learn subsequent instructions.

We note that the top-1 opcode probability grows with the sequence length s. This is reasonable, as we are effectively providing more contextual information. Considering Image Editing, moving from s=8 to s=16 increases the number of sequences with a top-1 opcode probability of 100%, from 87 to 94.

3.2 Learning Semantic Correlations in Machine Language

A straw-man approach is to store all instruction sequences and their top-1 opcodes in a look-up table. Although modern processors already store various runtime information (e.g., the branch history) in the tabular format, doing so in our case can be costly. First, such a table can take up a significant amount of spaces — for example, the RISC-V instruction set has N=200+ opcodes [2], and storing all s=8 sequences would need up to $200^8 \approx 2.56 \times 10^{18}$ table entries. The space complexity is ($O(N^s)$), as compared to typical branching history tables' $O(\#\ branches)$. Even running the smallest benchmark in SPEC-CPU-2017, Fluid-Dynamics (with ∼1,000 lines of C) already has about 12,625 s=8 sequences (or 26,748 s=16 sequences). Furthermore, there is a need to implement efficient look-ups for keys of length s, which is longer than the single program counter in typical branching history tables.

The use of language models. Language models offer a different paradigm to learning machine language, and it can be a more compact and efficient tool for capturing semantic correlations. We find that our problem matches well with neural language models [4], and the opcodes here can be viewed

as words in language modeling. Although language models only learn the conditional distribution of words, the observation that machine language exhibits a high top-1 opcode probability (c.f. Fig 1) implies that we can directly output the opcode. To this end, our effort is called *LM-ML* (language model for machine language).

LM-ML opts Transformer[24] for the neural language model. This is motivated by the observation that machine language exhibits a similar long-range interactions, to words in NLP. An example is where a nested branch is related to the upper-level branch that might be some instructions away. With Transformer, LM-ML is able to consider the relationship among instructions.

LM-ML's language model is a Transformer encoder of 6 layers, each with 8 heads. And we use an end-to-end input embedding layer, with embedding size d=512. The training is as follows. Similar to the practice for natural language corpus, we divide the instruction trace into fixed-length blocks of k instructions. Our current implementation uses k=512. Each block is represented by a k-dimensional vector of embeddings of k instructions' opcodes, i.e., $(E(I_1), ..., E(I_k))$. The model takes this vector as input. Its output is a $k \times N$ matrix, which has the probability distribution of all N possible opcodes, for each instruction in $(I_2, ..., I_{k+1})$. Then, we use *CrossEntropy* as the loss function, to consider the difference between the ground truth and the model output.

During inference, LM-ML takes in the vector of embeddings of s=16 instructions' opcodes. Then, based on the model's output of probability distribution for I_{s+1}, it selects the most probable opcode as the prediction. We note that this process can be extended beyond one subsequent instruction.

4 Implementation

This section describes how LM-ML's language model can be integrated with processor's branch predictor. The key idea is to take a branch instruction's subsequent instruction predicted by the language model, and compare with its branch-taken and branch-not-taken instructions. The details are shown in line 5–10 in Alg 1.

Our implementation takes advantage of the following processor features. First, processors already maintain a BTB (branch target buffer), which caches a branch's branch-taken program counter. We extend BTB for BTOB (branch target and opcode buffer), to include the opcode (c.f. line 4 in Alg 1). Second, since processors' instruction-cache stores a line worth of consecutive instructions, we can peek the branch-taken opcode (c.f. line 2–3 in Alg 1). In the rare case that BTOB and instruction-cache do not hold the information that LM-ML needs, we resort to other predictors such as Bimodal (c.f. line 11–13 in Alg 1). This is also applicable to cases where branch-taken and branch-not-taken instructions are the same (c.f. line 14–16 in Alg 1).

5 Evaluation

We evaluate the performance of LM-ML, and the resulting improvement that LM-ML brings to branch prediction and processors. Experiments are carried out with six benchmarks from SPEC-CPU-2017 [23]; these benchmarks are based on real-world applications, which include file compression, image manipulation, route planning, 3D rendering, chess game AI, and go game AI. In order to exercise different code paths, we try different inputs for these benchmarks if possible.

The experiment setup is as follows. We start by compiling all benchmarks for the popular RISC-V processor instruction set [1]. They are then simulated in Spike [3], a RISC-V simulator, to get traces of executed instructions. For every benchmark, we combine all traces and then divide them into blocks of 1-million (2^{20}) instructions. Traces are uniformly sampled — for every 10 blocks, we use 8 blocks for training the language model, 1 block for validation, and 1 block for testing. Finally, we use the popular ChampSim simulator [13], which takes in Spike's execution traces as inputs, to compute branch predictors' accuracy and processor IPC. Our comparison baselines are branch predictors employed in modern processors: heuristics-based Bimodal [22] and learning-based Perceptron [14].

Branch prediction performance. Metrics here are branch prediction accuracy (i.e., the percentage of branches whose outcome is correctly predicted), and MPKI (mis-predictions per kilo-instructions).

Table 1 shows that LM-ML achieves a branch prediction up to 11.03% and 8.87% higher than the Bimodal and Perceptron baselines, respectively. For benchmarks where the baselines already achieve

Algorithm 1: Branch prediction with LM-ML's instruction opcode prediction

Input: S = (I_1, ..., I_{s-1}, I_{branch}), PC = branch's program counter, M = trained model
Output: Boolean to indicate whether branch will be taken

```
// Predict the subsequent instruction opcode
pred_opcode := M.predict(S)
// Get the branch-taken and branch-not-taken instructions
size = getNumBytes(PC)
br_not_taken_opcode = instruction_cache.fetchOpcode(PC + size)
br_taken_opcode = BTOB.fetchOpcode(PC)
// Get the branch-taken and branch-not-taken instructions
if pred_opcode == br_taken_opcode then
    return TRUE
end
if pred_opcode == br_not_taken_opcode then
    return FALSE
end
// If LM-ML fails, fall back to the default predictor such as Bimodal
if (br_not_taken_opcode == NULL) or (br_taken_opcode == NULL) then
    return defaultBranchPredictor()
end
if br_not_taken_opcode == br_taken_opcode then
    return defaultBranchPredictor()
end
```

Table 1: Branch prediction accuracy of LM-ML and baselines. The higher value is better.

Benchmark (*type*)	Input	LM-ML	Bimodal (*vs. LM-ML*)	Perceptron (*vs. LM-ML*)
mcf (*Route planning*)	400 trips	97.19%	96.19% (*-1.00%*)	96.78% (*-0.41%*)
mcf (*Route planning*)	3,500 trips	88.38%	83.60% (*-4.78%*)	86.54% (*-1.84%*)
mcf (*Route planning*)	9,000 trips	87.23%	83.32% (*-3.91%*)	86.35% (*-0.88%*)
xz (*Compression*)	Doc (1.3 MB)	93.55%	89.37% (*-4.18%*)	90.62% (*-2.93%*)
xz (*Compression*)	Mixed (7.9 MB)	90.09%	86.93% (*-3.16%*)	87.88% (*-2.21%*)
xz (*Compression*)	Code (0.2 MB)	94.52%	90.88% (*-3.64%*)	91.96% (*-2.56%*)
xz (*Compression*)	PDF (1 MB)	96.13%	91.35% (*-4.78%*)	92.88% (*-3.25%*)
imagick (*Image editing*)	10×10 images	99.46%	92.73% (*-6.73%*)	96.32% (*-3.14%*)
blender (*3D rendering*)	WCG dataset (cube)	97.71%	93.48% (*-4.23%*)	95.11% (*-2.60%*)
leela (*Go game AI*)	Game 1	90.72%	79.69% (*-11.03%*)	81.85% (*-8.87%*)
leela (*Go game AI*)	Game 2	89.31%	80.63% (*-8.68%*)	82.40% (*-6.91%*)
deepsjeng (*Chess game AI*)	Game 1	94.49%	93.07% (*-1.42%*)	94.24% (*-0.25%*)
deepsjeng (*Chess game AI*)	Game 2	95.36%	90.06% (*-5.30%*)	91.66% (*-3.70%*)
deepsjeng (*Chess game AI*)	Game 3	92.74%	86.86% (*-5.88%*)	88.92% (*-3.82%*)

a fairly high prediction accuracy, LM-ML is still able to outperform Bimodal and Perceptron by at least 1.00% and 0.25%, respectively. We delve into Table 1 with the following observations. First, LM-ML's higher branch prediction accuracy is due to the capability to better infer upcoming instructions — considering the case of predicting upcoming instructions for Image Editing, LM-ML can achieve an average of 99.58% accuracy in predicting the next instruction (and even an average of 97.30% in predicting the next 16 instructions). Second, since we use one language model for each benchmark, the semantic correlations learned is transferable among different benchmark inputs.

Table 2 shows the MPKI comparisons. Unlike prediction accuracy, MPKI quantifies the impact of branch predictions on program execution. For every 1,000 instructions executed, LM-ML reduces the number of branch mis-predictions by an average of 87.49% and 50.37%, as compared to the Bimodal and Perceptron baselines. The table also shows a general trend: the Perceptron tends to outperform Bimodal, and language model can further improve the performance.

Processor performance. We quantify the improvement that LM-ML's better branch prediction brings to the processor performance, in terms of IPC (instructions executed per cycle). The ChampSim

Table 2: Branch prediction MPKI of LM-ML and baselines. The lower value is better.

Benchmark (*type*)	Input	LM-ML	Bimodal (*vs. LM-ML*)	Perceptron (*vs. LM-ML*)
mcf (*Route planning*)	400 trips	6.36	8.62 (*+35.53%*)	7.29 (*+14.62%*)
mcf (*Route planning*)	3,500 trips	25.39	35.84 (*+41.16%*)	29.41 (*+15.83%*)
mcf (*Route planning*)	9,000 trips	27.85	36.39 (*+30.66%*)	29.79 (*+6.97%*)
xz (*Compression*)	Doc (1.3 MB)	6.13	9.88 (*+61.17%*)	8.71 (*+42.09%*)
xz (*Compression*)	Mixed (7.9 MB)	11.32	14.93 (*+31.89%*)	13.84 (*+22.26%*)
xz (*Compression*)	Code (0.2 MB)	5.90	9.83 (*+66.61%*)	8.67 (*+46.95%*)
xz (*Compression*)	PDF (1MB)	3.85	8.61 (*+123.64%*)	7.08 (*+83.90%*)
imagick (*Image editing*)	10×10 images	2.71	9.47 (*+249.44%*)	4.79 (*+76.75%*)
blender (*3D rendering*)	WCG dataset (cube)	2.82	8.02 (*+184.40%*)	6.02 (*+113.48%*)
leela (*Go game AI*)	Game 1	7.81	17.07 (*+118.57%*)	15.21 (*+94.75%*)
leela (*Go game AI*)	Game 2	9.32	16.90 (*+81.33%*)	15.35 (*+64.70%*)
deepsjeng (*Chess game AI*)	Game 1	7.61	9.58 (*+25.89%*)	7.96 (*+4.60%*)
deepsjeng (*Chess game AI*)	Game 2	5.54	11.88 (*+114.44%*)	9.97 (*+79.96%*)
deepsjeng (*Chess game AI*)	Game 3	7.53	13.63 (*+81.01%*)	11.49 (*+52.59%*)

Table 3: Processor IPC of LM-ML and baselines. The higher value is better.

Benchmark (*type*)	Input	LM-ML	Bimodal (*vs. LM-ML*)	Perceptron (*vs. LM-ML*)
mcf (*Route planning*)	400 trips	2.850	2.648 (*-7.07%*)	2.763 (*-3.04%*)
mcf (*Route planning*)	3,500 trips	2.067	1.766 (*-14.58%*)	1.940 (*-6.16%*)
mcf (*Route planning*)	9,000 trips	1.986	1.746 (*-12.09%*)	1.926 (*-3.03%*)
xz (*Compression*)	Doc (1.3 MB)	2.671	2.325 (*-12.96%*)	2.423 (*-9.29%*)
xz (*Compression*)	Mixed (7.9 MB)	2.136	1.890 (*-11.50%*)	1.958 (*-8.32%*)
xz (*Compression*)	Code (0.2 MB)	2.924	2.508 (*-14.24%*)	2.618 (*-10.47%*)
xz (*Compression*)	PDF (1 MB)	2.683	2.352 (*-12.32%*)	2.449 (*-8.71%*)
imagick (*Image editing*)	10×10 images	3.579	2.867 (*-19.90%*)	3.208 (*-10.37%*)
blender (*3D rendering*)	WCG dataset (cube)	3.292	2.702 (*-17.92%*)	2.902 (*-11.84%*)
leela (*Go game AI*)	Game 1	2.486	1.960 (*-21.16%*)	2.047 (*-17.66%*)
leela (*Go game AI*)	Game 2	2.374	1.970 (*-17.01%*)	2.041 (*-14.02%*)
deepsjeng (*Chess game AI*)	Game 1	2.572	2.417 (*-6.03%*)	2.543 (*-1.13%*)
deepsjeng (*Chess game AI*)	Game 2	2.788	2.259 (*-18.98%*)	2.396 (*-14.07%*)
deepsjeng (*Chess game AI*)	Game 3	2.557	2.109 (*-17.51%*)	2.247 (*-12.11%*)

simulator can estimate the IPC for comparison baselines, which we then use to estimate the IPC for LM-ML. Table 3 shows that LM-ML can improve the processor IPC by up to 21.16% and 17.66%, as compared to Bimodal and Perceptron. An interesting observation is that even a small improvement in branch prediction can translate into a significant processor performance improvement. This is due to the cost of recovering from branch mis-predictions.

6 Conclusion and Future Work

This paper takes the first step, towards learning the semantic correlation in machine language. Empirical results demonstrate the significant gain in instruction-related speculations (e.g., branch predictions) and processor performance, with SPEC-CPU-2017 benchmarks.

As an ongoing effort, we are addressing the inference overhead of our language model, especially that a large overhead can cause processors to stall when making speculations. First, we are exploring compression techniques to reduce the model size. Second, we are exploring how processors might better use language models by predicting a sequence of upcoming instruction opcodes, rather than only one instruction. Finally, due to the limited expressiveness of individual processor instruction, it is possible that LM-ML needs to increase the sequence length s for complex programs such as databases. We are evaluating how language models would behave with long input sequences.

References

[1] RISC-V GNU Compiler Toolchain. `http://github.com/riscv-collab/riscv-gnu-toolchain`.

[2] RISC-V Instruction Set. `http://riscv.org/specifications/privileged-isa`.

[3] Spike: RISC-V ISA Simulator. `http://github.com/riscv-software-src/riscv-isa-sim`.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 2003.

[5] Pavol Bielik, Veselin Raychev, and Martin Vechev. PHOG: Probabilistic Model for Code. In *ICML*, 2016.

[6] Mercel Bruch, Martin Monperrus, and Mira Mezini. Learning from Examples to Improve Code Completion Systems. In *ESEC/FSE*, 2009.

[7] Ioana Burcea and Andreas Moshovos. Phantom-BTB: A Virtualized Branch Target Buffer Design. In *ASPLOS*, 2009.

[8] I-Cheng K. Chen, Chih-Chieh Lee, and T.N. Mudge. Instruction Prefetching using Branch Prediction Information. In *ICCD*. IEEE, 1997.

[9] Stijn Eyerman, James E. Smith, and Lieven Eeckhout. Characterizing the Branch Misprediction Penalty. In *ISPASS*, 2006.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*. ACL, 2020.

[11] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive Instruction Fetch. In *MICRO*, 2011.

[12] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A. Jiménez. Bit-level Perceptron Prediction for Indirect Branches. In *ISCA*, 2019.

[13] Intel. ChampSim. `https://github.com/ChampSim/ChampSim`.

[14] Daniel A. Jimenez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *HPCA*, 2001.

[15] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *ISCA*, 2021.

[16] Chit-Kwan Lin and Stephen J. Tarsa. Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions. In *IISWC*, 2019.

[17] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An Imitation Learning Approach for Cache Replacement. In *ICML*, 2020.

[18] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *PLDI*. ACM, 2014.

[19] Alberto Ros and Alexandra Jimborean. A Cost-Effective Entangling Prefetcher for Instructions. In *ISCA*, 2021.

[20] André Seznec and Pierre Michaud. A Case for (Partially) TAgged GEometric History Length Branch Prediction. *Journal of Instruction-level Parallelism*, 2006.

[21] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying Deep Learning to the Cache Replacement Problem. In *MICRO*, 2019.

[22] James E. Smith. A Study of Branch Prediction Strategies. In *ISCA*, 1981.

[23] Standard Performance Evaluation Corporation (SPEC). SPEC CPU 2017. https://www.spec.org/cpu2017.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *NIPS*, 2017.

[25] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt. BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches. In *MICRO*, 2020.

[26] Anastasios Zouzias, Kleovoulos Kalaitzidis, and Boris Grot. Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies. In *MLArchSys*. ACM, 2021.