Final Project : Branch Predictor with LSTM & Attention Mechanism

Team members: 吳宥毅 109550128, 許登豪 109550175, 楊立嘉 109550053

Abstract

We implement a branch predictor integrating both neural networks composed of several LSTM layers and attention mechanism as a reference for most of the branch, alongside with a loop predictor implemented in TAGE-SC-L. We compare our performance with that of TAGE-SC-L and 2 bit FSM under several interesting PC reference patterns. Although the result shows that our design generally outperforms simple 2 bit FSM, and still loses to TAGE-SC-L, it shows us the research potential of neural network-based branch predictor.

Introduction

Branch predictor is a CPU component that improves the flow of instruction execution by guessing the outcome of conditional branch instructions before the branch outcome is resolved in later pipeline stages. We can see it as a task that a binary classifier can complete. Moreover, we are taught in class that some research has been done about utilizing perceptron to serve as a predictor or neural network to choose between local and global predictor. Seeing the 2 points mentioned above as inspiration and aiming to surpass common counter-based branch prediction mechanisms to achieve better results, we try to use the machine learning method as a branch predictor and evaluate its effectiveness.

We will utilize techniques derived from natural language processing : For example substituting the concept of a “token” with the Program Counter Address, incorporate not only linear layers but also bidirectional LSTM layers, attention mechanism, etc. This model will be trained online to accommodate a possibly changing working set. Also, recognizing that loops are a regular and prominent pattern in contemporary workloads, we will also integrate a loop predictor, as implemented in TAGE-SC-L, alongside with our proposed inference model. Our results will be compared using several traces against a 2-bit FSM and the current state-of-the-art method, TAGE-SC-L. The comparison will focus on aspects such as working set learning capability and speed, total bits usage, and tolerance to changing patterns.

Implementation

1. Overview of our binary classifier model

- Input : concat past PC history + Current PC + Speculative PC together
- Output : a value between 0 and 1, indicating whether this sequence is valid
- When to update : we use online update, in order to be adaptive to changing reference pattern

2. Important Functions

```
GetPrediction(
   UINT64 PC (where we need to predict),
   UINT64 Speculative PC (if the branch is taken where we will go to)
   ):
      bool predloop = get_prediction_by_loop_predictor(PC)
      bool prednet = get_prediction_by_net(PC, Speculative_PC)
      if (loop_condition_satisfy) : return predloop
      else : return prednet
```

```
UpdatePrediction(
    UINT64 PC (where we need to predict),
    UINT64 Speculative PC (if the branch is taken where we will go to),
    bool resolveDir (the correct prediction),
    bool predDir (the prediction we make)
    ):
        update_loop_predictor(PC, resolveDir, predDir, Speculative PC)
        # update loop predictor unconditionally

        record predicting result of network and ground truth
        if (not enough prediction for a batch update):
            return
        else: update_net(resolveDir, predDir)
        # do backpropagation here
```

```
get_prediction_by_net(UINT64 PC, UINT64 Speculative_PC):

    # We will omit some code here:
    # Divide PC and Speculative_PC into chunks in 16 bits (ex: for a PC of 64 bits, we
    # will divide it into 4*16 bits), to save embedding layer memory usage
    # Concatenate previous PC history(also in chunks and stored beforehand) + PC in
    # chunks + Speculative_PC in chunks into PCs_chunks_tensor

    network.train()

    # no matter this prediction is made by loop predictor or not, we train the model
    branch_taken_probability = network(PCs_chunks_tensor)
    prednet = bool(branch_taken_probability[0] > 0.5)
    return prednet
```

3. Network

- Network Structure

```
AI_Predictor(
    (embeddings_1): Embedding(65536, 100)
    (embeddings_2): Embedding(65536, 100)
    (embeddings_3): Embedding(65536, 100)
    (embeddings_4): Embedding(65536, 100)
    (lstm): LSTM(400, 128, batch_first=True, bidirectional=True)
    (relu): ReLU()
    (fc1): Linear(in_features=256, out_features=256, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (fc2): Linear(in_features=256, out_features=1, bias=True)
    (sigmoid): Sigmoid()
    (attention_layer): Attention(
        (attn): Linear(in_features=512, out_features=256, bias=True)
        (v): Linear(in_features=256, out_features=1, bias=True)
    )
)
```

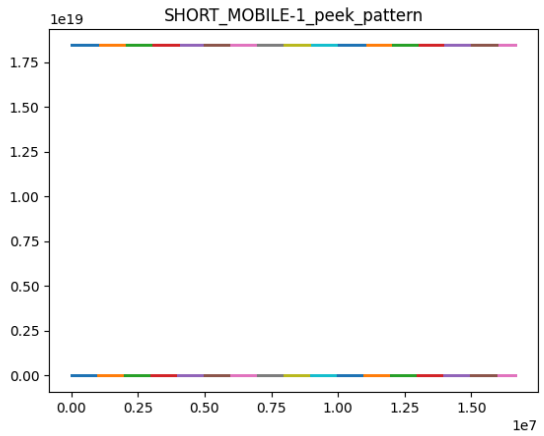
- Network loss function and optimizer

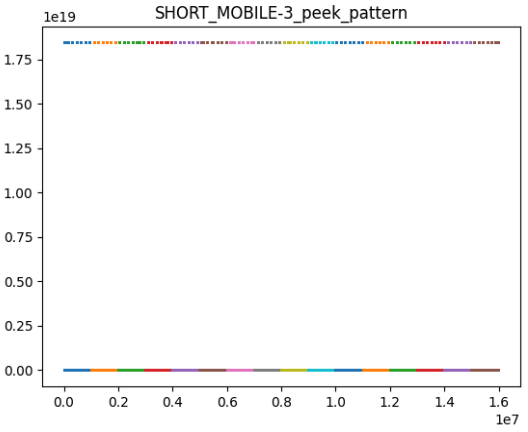
```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(network.parameters(), lr=1e-3)
```

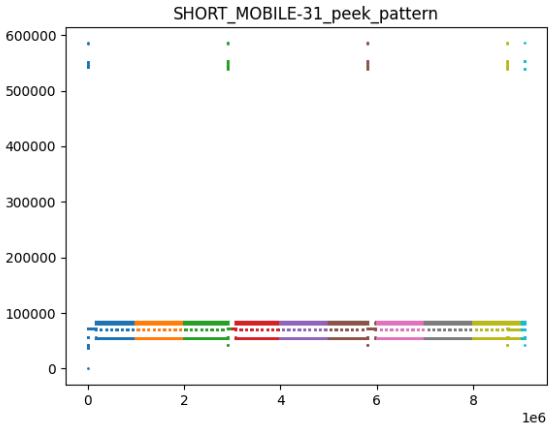
Experiment Setup

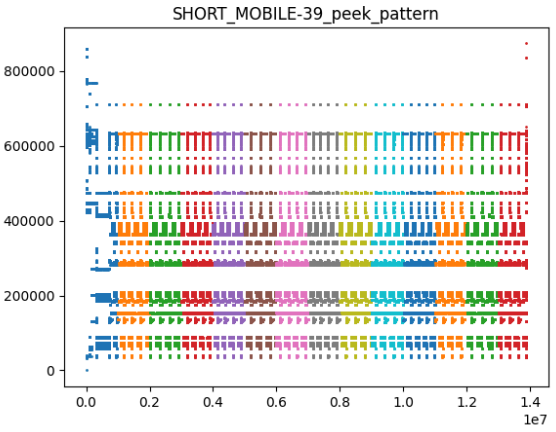
1. Trace - SHORT_MOBILE (figure : time v.s. PC address)

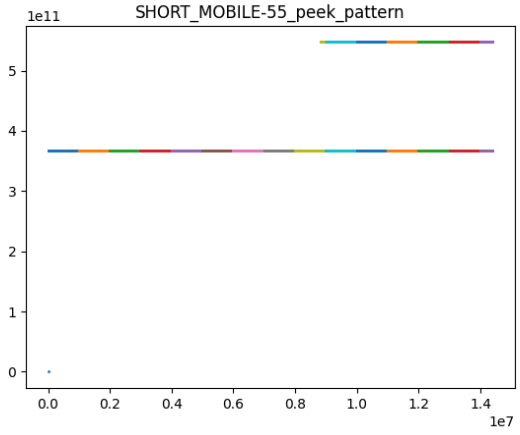
Due to the tight schedule at the end of the semester and after printing all the traces, we found that most of the traces have similar patterns. Therefore, we manually selected a few representative traces that are more valuable for discussion : SM(abbreviation for SHORT_MOBILE)-1 has 2 dense working sets, SM-3 also has 2 dense sets but one of it is sparsely referenced, SM-31 has 1 broad set and some floating referenced address, SM-39 has wonderful temporal and spatial locality just as the picture on text book, SM-55 has 2 dense sets but one shows abruptly in the later stage, SM-61 has 4 dense working sets.

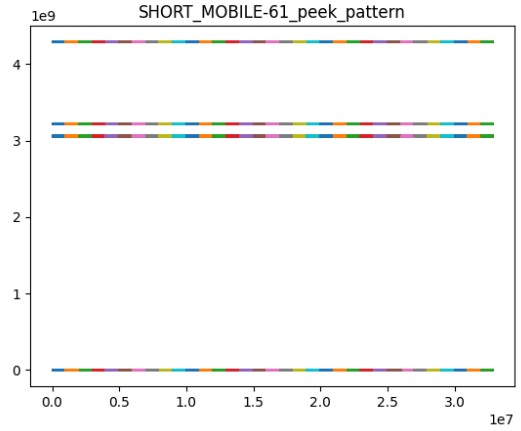












2. Environment

- OS : Ubuntu 22.04.4 LTS
- GPU : NVIDIA GeForce RTX 3090
- Python : 3.10.12
- torch : 2.1.1+cu118

3. Benchmark

1. Simple BHT composed of 2-bits FSM predictors
   We implemented a 2-bit FSM predictor with a Branch History Table consisting of 1024 entries.
2. Tage-SC-L predictor
   We use the Tage-SC-L which is the default implementation of the contest.

4. Metrics

- performance - MISPRED_PER_1K_INST
- # bits used to record history

Experiment Result & Analysis

1. Performance - MISPRED_PER_1K_INST

The table below is the mispredict count per 1K instruction under different traces and models. Our method wins 2-bit FSM but loses to TAGE-SC-L almost all the time.

	trace 1	trace 3	trace 31	trace 39	trace 55	trace 61
2 bit FSM	4.2141	4.7015	0.9286	2.1687	10.7166	10.0561
Our Model	3.8931	4.7777	0.1321	1.5477	11.2219	4.6990
TAGE-SC-L	0.3210	0.2002	0.0001	0.9927	8.4770	0.4888

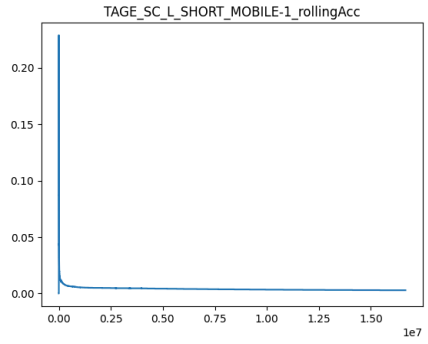
2. # bits used to record history

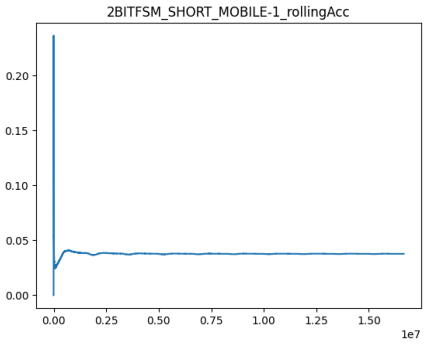
Below are a number of bits that are used in three models to record history. Our method uses a little more bits than the 2-bit FSM and thus achieves better performance. And we guess that why we lose to TAGC-SC-L are the following 2 reasons : (A) we only use 500 past PCs to help us determine the result, our result might be better if we increase it. (B) TAGE-SC-L uses arrays to record groundtruth and its prediction of past branches, and it also uses a SC to calibrate its prediction, while our method only obscurely turns each misprediction as some parts of loss value.

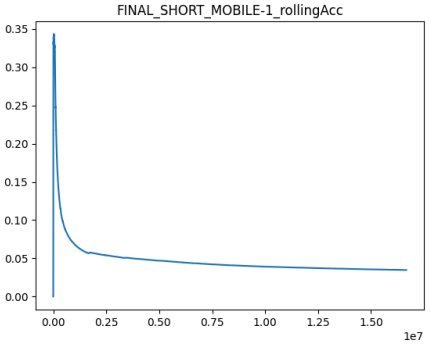
- 2 bit FSM : 1024 entries per Branch History Table * 2 bits to represent a state per BHT entry = 2K bits
- Our Model : 500 past PCs * 64 bits per PC + LOOP 1248 bits = 33248 bits
- TAGE-SC-L : 523355 bits(TAGE 463917 bits + LOOP 1248 bits + SC 58190 bits

3. Warm up time

These figures show the number of branches currently executed v.s. # misprediction currently made divided by branches currently executed. The name “rollingAcc” might be confusing. We originally thought that our model may take more time to reach its stable stage, but the results turned out that it is still acceptable compared to TAGE-SC-L and 2 bit FSM. Why the warm up time of 2 bit FSM is faster is because this FSM has only 4 stages, while our model needs to update thousands of neurons to alter the prediction.

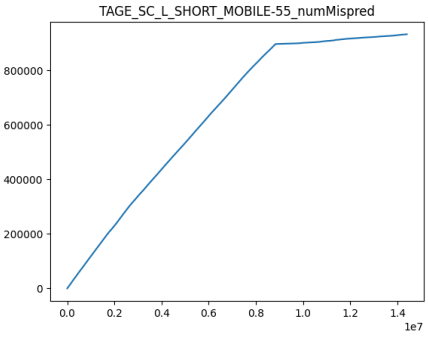


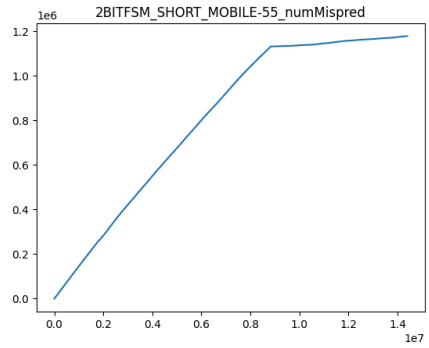


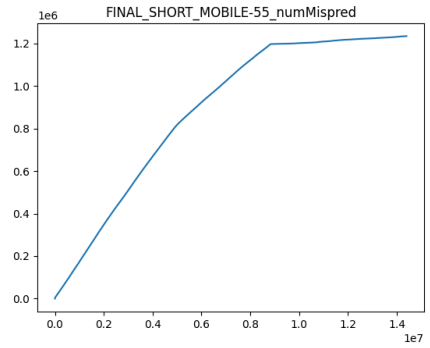


4. Tolerance to changing patterns

We can see that for trace SM-55, it contains 2 dense sets but one shows abruptly in the later stage. So this is a good case for us to test the tolerance to changing patterns with different algorithms. From the slope starting from the inflection point (this is just when the new working set appears) in each graph below we can see that our model cannot adapt to the new working set as quickly as that of TAGE-SC-L. Please note that although they appear very similar in each figure, due to the difference in the y-axis scale, it can be observed that the slope of our model is steeper. This suggests that neural network-based branch predictor may not be able to respond in a timely manner when encountering newly emerging working sets.







5. Future work

In the future work, we plan to explore network size reduction and early-stopping techniques to improve reference speed.

Related Work

[1] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Monterrey, Mexico, 2001, pp. 197-206, doi: 10.1109/HPCA.2001.903263.

This paper points out that the key concept behind 2-bit Finite State Machine (FSM), which transitions between states based on both predicted and actual outcomes, is akin to the concept behind how perceptrons work. This inspires us that we can use the AI model as a form of branch predictor and try to gain better performance through it.

[2] Seznec, André. (2014). TAGE-SC-L Branch Predictors.

TAGE-SC-L is the current SOTA branch predicting algorithm that does not involve fancy machine learning related methods. It inherits TAGE and adds a LOOP predictor. Everytime the branch predictor is referenced it either consults to TAGE or LOOP predictor, and then alters the prediction if needed via SC (statistical corrector). Since our goal is to try to use our model to surpass TAGE-SC-L, we compare our result with it.