ADVANCE COMPUTER ARCHITECHTURE
CS G524

BITS Pilani
K K Birla Goa Campus

FINAL PROJECT REPORT

To Dr. Kanchan Manna

From Sai Charan, Aditya Agarwal, Yogesh Agrawal – Group 12

Date Apr 23, 2025

Subject: Milestone 2 report – Branch prediction algorithms

Branch prediction algorithms - Evaluation of Traditional and ML-Inspired Branch Prediction Algorithms

1. Introduction

Branch prediction remains a critical component in modern high-performance processors, directly influencing instruction throughput and overall efficiency by mitigating the penalties associated with control flow changes. While traditional hardware predictors like TAGE and Perceptron variants have achieved high accuracy levels, they face diminishing returns, particularly with complex branch patterns dependent on long or noisy history. Recent research has explored alternative paradigms, including Machine Learning (ML) and Reinforcement Learning (RL), framing branch prediction as a learning problem. Notably, work by Zouzias et al. [4] proposed an RL framework, while Vontzalidis [3], building on Tarsa et al., developed "BranchNet," a Convolutional Neural Network (CNN) approach targeting branches that are "Hard-to-Predict" (H2P) for conventional predictors.

This project initially aimed to replicate and evaluate key aspects of these advanced ML/RL approaches within a simulation environment. However, due to challenges encountered with legacy codebases and dependencies, the focus shifted towards developing a simplified Python-based simulation framework. This framework allows for the direct comparison of fundamental traditional branch predictors against simplified, yet conceptually representative, ML models inspired by BranchNet, using synthesized branch history data designed to challenge different prediction strategies.

This simulation of branch prediction algorithms is mainly inspired from work of [3],[7] and [8]

2. Project Overview and Objectives

The primary objective of this project was to explore and evaluate the potential of machine learning techniques in improving branch prediction accuracy compared to established hardware algorithms.

The project involved the following high-level steps:

1. Initial Research & Replication Attempt: Study the concepts presented in [3, 4] and attempt to set up the associated code repositories and the ChampSim simulation environment for reproducing published results.
2. Methodology Pivot: Due to technical challenges with legacy software, transition to a self-contained Python simulation environment.
3. Data Synthesis: Develop a methodology to generate synthetic, yet challenging, branch history sequences representative of patterns that might trouble traditional predictors.
4. Predictor Implementation: Implement standard baseline predictors (1-Bit, 2-Bit, 2-Level, Tournament) in Python.
5. ML Model Development: Implement simplified ML models (CNN, CNN+LSTM, Transformer Encoder) using Keras/TensorFlow, capturing the architectural essence of predictors discussed in [3].
6. Simulation & Evaluation: Simulate all implemented predictors against the synthesized branch history, measuring prediction accuracy.

7. Visualization & Analysis: Generate comparative visualizations of predictor performance and analyze the results.
8. Web-based Reporting: Develop a simple Flask web application to present the project findings, methodology, and results in an accessible format.

3. Initial Approach and Challenges: Replicating Research Results

The project initially intended to directly replicate experiments from the Vontzalidis thesis repository [3] and potentially explore RL concepts from Zouzias et al. [4] using the ChampSim simulator.

- ChampSim & Trace Files: ChampSim is a trace-based architectural simulator. The standard workflow involves generating detailed execution traces using instrumentation tools like Intel Pin. These traces record information about executed instructions, memory accesses, and crucially, branch outcomes (PC and Taken/Not-Taken status). ChampSim reads these traces and simulates the behavior of processor components, including the branch predictor, allowing for metrics like MPKI (Mispredictions Per Kilo-Instructions) and IPC (Instructions Per Cycle) to be measured.
- Hard-to-Predict (H2P) Branches: The BranchNet methodology [3] focuses on identifying H2P branches – specific static branches in a program that conventional predictors (like TAGE) consistently mispredict with high frequency. The process typically involves:
    1. Running a benchmark trace through a baseline predictor simulation (e.g., TAGE within ChampSim).
    2. Collecting statistics (frequency, misprediction count, accuracy) for each static branch PC.
    3. Filtering branches based on predefined criteria (e.g., >15k executions, >1k mispredictions, <99% accuracy) to identify the H2P set.
    - The idea is to train specialized, computationally expensive ML models *only* for these few problematic H2P branches offline and use a standard predictor for the rest at runtime.
- Repository Challenges: Significant hurdles were encountered when attempting to set up and run the code from [3]:
    - Dependency Conflicts: The repository relied on specific, older versions of libraries (e.g., Python 3.8, PyTorch 1.3.0, specific Protobuf versions). Reconciling these with modern development environments and operating system repositories proved difficult, leading to installation failures and runtime errors (as documented in interaction logs).
    - Simulator Build Issues: The extended version of ChampSim recommended in the repository failed to compile successfully in the available VMware virtual machine environment, potentially due to toolchain differences or missing dependencies specific to the author's original setup. Official ChampSim has also evolved significantly since the version likely used for the thesis.
    - Complexity: The provided training and evaluation scripts (branchnetTrainer.py, BatchDict_testerAll.py) were acknowledged in the repository's README as complex and tightly coupled to the author's workflow, requiring substantial effort to adapt and debug.

Given these obstacles, a strategic decision was made to pivot away from direct replication and focus on simulating the *core concepts* in a controlled Python environment. This allowed for exploration of the relative performance of different predictor philosophies without being blocked by legacy software issues.

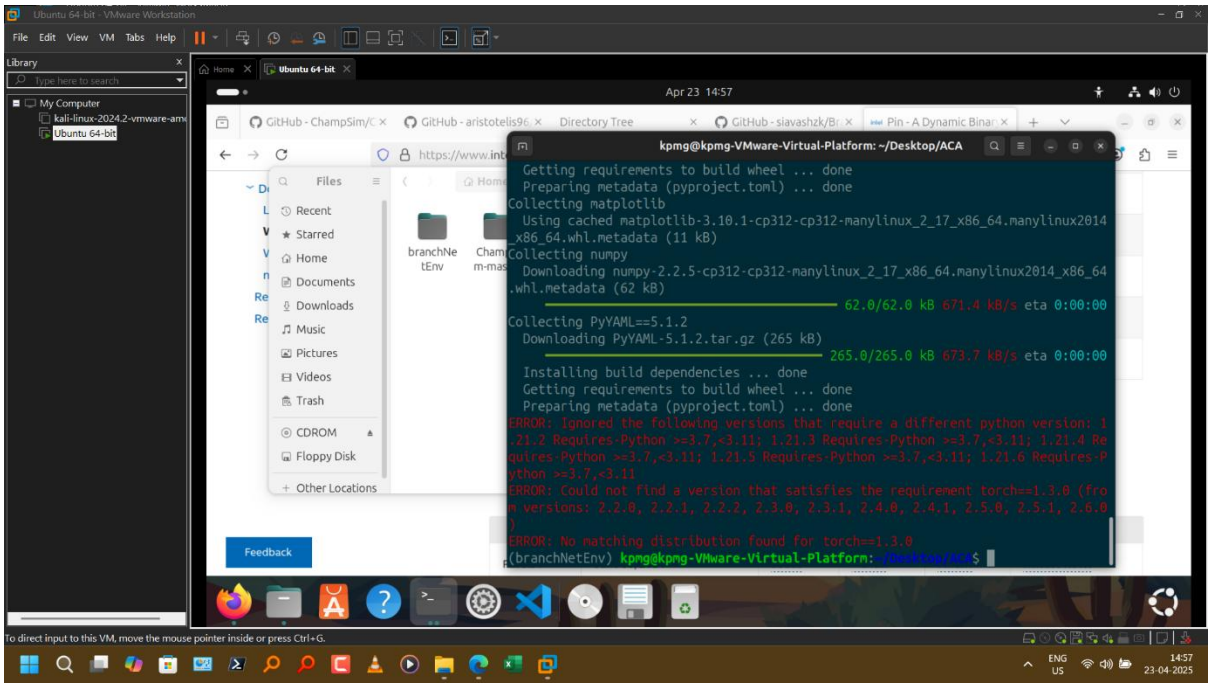

Fig 1: Issues with legacy implementations of the research

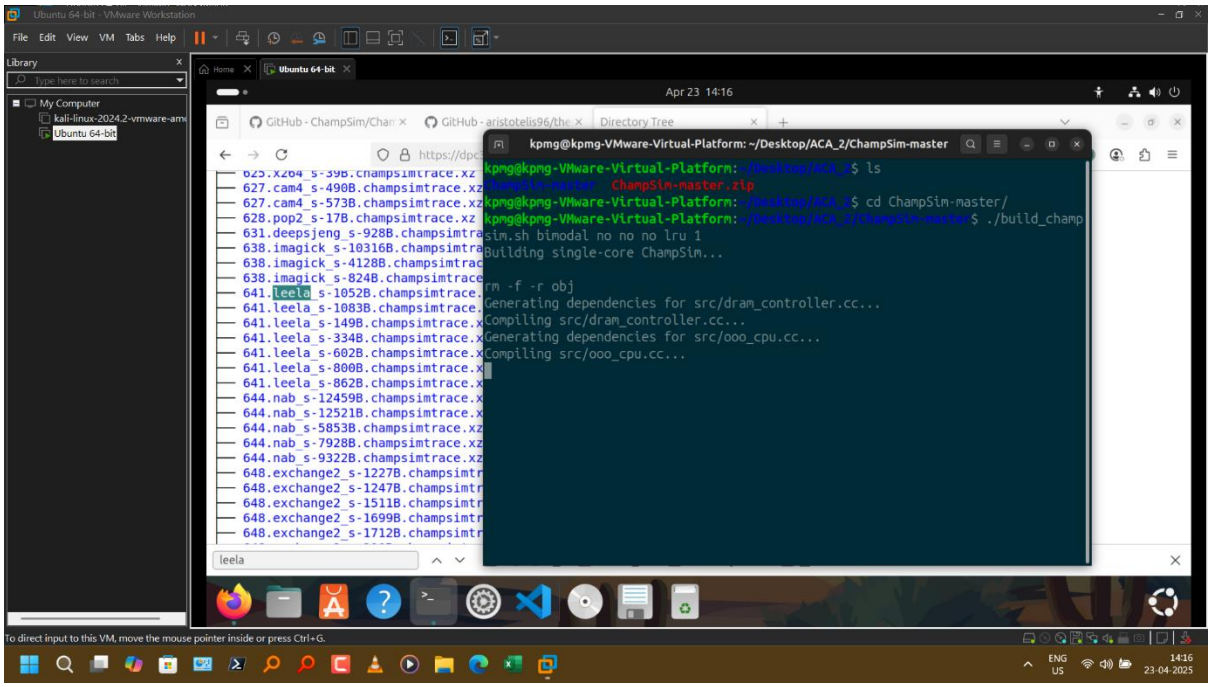

Fig 2: ChampSim simulator has changed considerably after the version which supported original branch net and related code

4. Simulation Framework: Dataset, Code, and Logic

To facilitate comparison, a simplified simulation environment was created in Python:

Our code is available at: [NeuroNut/Branch-prediction-algorithms-Adv-Computer-Architechture: Basic and simplified simulation and evaluation of various BP algos]

- Dataset Generation (generate_data.py):
    - A function (generate_branch_history_complex) was developed to synthesize a sequence of branch outcomes (0 for Not Taken, 1 for Taken).
    - The outcome logic was designed to be challenging: Outcome = 1 if (i % 7 == 0) XOR (history[i-5] == 0) else 0. This incorporates periodicity (i % 7) and a specific historical dependency (history[i-5]) combined with non-linear XOR logic. An option for adding random noise was included but disabled (noise=0) for the final reported run to focus on deterministic predictability.
    - A helper function (create_windows) transforms the generated history sequence into overlapping windows suitable for training sequence-based ML models (Input: window of N previous outcomes, Output: the next outcome).
    - The generated history and windowed data (X, y) were saved to a .npz file for use by the simulation script.
- Predictor Implementations (simulate_predictors.py):
    - Basic Predictors: Classes were implemented for:
        - OneBitPredictor: Maintains a single state bit (Predict T/NT), flips on mispredict.
        - TwoBitPredictor: Uses a 2-bit saturating counter (Strongly NT, Weakly NT, Weakly T, Strongly T).
        - TwoLevelPredictor: Implements a simple Gshare-like predictor using a Global History Register (GHR, simulated as a deque) XORed with the PC to index a Pattern History Table (PHT) of 2-bit saturating counters.
        - TournamentPredictor: Combines a TwoBitPredictor (local) and a TwoLevelPredictor (global), using a selector table (indexed by PC) to choose which predictor's output to use based on their recent relative success.
    - ML Models (Keras/TensorFlow): Functions were created to build simplified Keras models inspired by BranchNet concepts:
        - build_simple_branchnet_cnn: A purely CNN model with multiple Conv1D layers, BatchNormalization, MaxPooling, and Dense layers, aiming to extract spatial features from the history window.
        - build_simple_brangnet_lstm: Combines initial Conv1D layers for feature extraction followed by an LSTM layer to explicitly model temporal sequences within the history window.
        - build_simple_branchnet_transformer: Uses Keras's MultiHeadAttention and feed-forward blocks to implement a basic Transformer Encoder layer, exploring attention mechanisms for identifying relevant parts of the history.

```
def build_simple_branchnet_cnn(history_window_size, num_features=1):

    model = keras.Sequential(
        [
```

```
            keras.Input(shape=(history_window_size, num_features)),
            layers.Conv1D(filters=32,    kernel_size=5,    padding='causal',
activation="relu"),
            layers.BatchNormalization(),
            layers.Conv1D(filters=64,    kernel_size=5,    padding='causal',
activation="relu"),
            layers.BatchNormalization(),
            layers.MaxPooling1D(pool_size=2), # Added Pooling
            layers.Conv1D(filters=64,    kernel_size=3,    padding='causal',
activation="relu"), # Added Layer
            layers.GlobalAveragePooling1D(),
            layers.Dense(32, activation="relu"),
            layers.Dropout(0.3),
            layers.Dense(1, activation="sigmoid"),
        ]
    )
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
    model.compile(loss="binary_crossentropy",            optimizer=optimizer,
metrics=["accuracy"])
    return model

def build_simple_branchnet_lstm(history_window_size, num_features=1):

    model = keras.Sequential(
        [
            keras.Input(shape=(history_window_size, num_features)),
            layers.Conv1D(filters=32,    kernel_size=5,    padding='causal',
activation="relu"),
            layers.BatchNormalization(),
            layers.Conv1D(filters=64,    kernel_size=5,    padding='causal',
activation="relu"),
            layers.BatchNormalization(),
            layers.LSTM(32, return_sequences=False),
            layers.Dense(32, activation="relu"),
            layers.Dropout(0.3),
            layers.Dense(1, activation="sigmoid"),
        ]
    )
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005)
    model.compile(loss="binary_crossentropy",            optimizer=optimizer,
metrics=["accuracy"])
    return model
```

- Simulation Logic (simulate_predictors_no_fusion):
    - Takes the full generated history sequence as input.
    - Initializes instances of all basic predictors and the trained ML models.
    - Iterates through the history sequence step-by-step.
    - In each step i:
        - Retrieves the actual_outcome from the history.
        - Calls the predict() method of each *basic predictor* based on their internal state derived from history *up to step i-1*.
        - Compares the prediction to the actual_outcome and updates accuracy counters.
        - Calls the predict() method of each *ML model*, feeding it the window of the last HISTORY_WINDOW_SIZE *actual outcomes* (from steps i-HISTORY_WINDOW_SIZE to i-1). Compares prediction and updates counters (only after the initial window fill).

        - Calls the update() method of each *basic predictor*, providing the actual_outcome from step i so they can update their internal state (counters, GHR) for the *next* prediction at step i+1.
        - Maintains a history_buffer (deque) of actual outcomes for feeding the ML models in the next iteration.
    - Calculates and returns the final accuracy for each predictor.

```
def simulate_predictors_no_fusion(full_history, basic_predictors, ml_models, history_window_size=16):
    """Simulates predictors and returns their accuracy. No Fusion."""
    # Initialize results dictionaries
    results = {name: {'correct': 0, 'total': 0} for name in basic_predictors}
    results.update({name: {'correct': 0, 'total': 0} for name in ml_models})

    branch_pc = 0xABC # Fixed PC

    history_buffer = deque([0] * history_window_size, maxlen=history_window_size) # For ML models

    # --- Performance Profiling Variables ---
    ml_predict_time = 0
    basic_predict_time = 0
    loop_start_time = time.time()
    # ---------------------------------------

    for i in range(len(full_history)):
        actual_outcome = full_history[i]
        can_predict_ml_this_step = (i >= history_window_size)

        # Predicting with Standalone Basic Predictors
        t_start = time.perf_counter()
        basic_preds_this_step = {}
        for name, predictor in basic_predictors.items():
            basic_preds_this_step[name] = predictor.predict(branch_pc) # Store prediction
            if basic_preds_this_step[name] == actual_outcome:
                results[name]['correct'] += 1
            results[name]['total'] += 1
        basic_predict_time += (time.perf_counter() - t_start)


        # Predicting with Standalone ML Models
        t_start = time.perf_counter()
        if can_predict_ml_this_step:
            current_window = np.array(history_buffer).reshape(1, history_window_size, 1)
            for name, model in ml_models.items():
                # Predict probability
                prob_taken = model.predict(current_window, verbose=0)[0][0]
                # Threshold probability
                ml_pred = 1 if prob_taken >= 0.5 else 0
```

```
                if ml_pred == actual_outcome: results[name]['correct']
+= 1
                results[name]['total'] += 1 # Total increments only
after warm-up
        ml_predict_time += (time.perf_counter() - t_start)


        # UPDATE Phase (Only Basic Predictors Need State Update)
        for name, predictor in basic_predictors.items():
            predictor.update(branch_pc, actual_outcome)


        # Update History Buffer for NEXT iteration's ML predictions
        history_buffer.append(actual_outcome)

        # Progress Indicator
        if (i + 1) % 100 == 0: # Print progress every 100 steps
             print(f"\rSimulating step {i+1}/{len(full_history)}...",
end="")
```

```
Generating 1000 complex branch outcomes (Noise=0.0%, Correlation Depth=5)...
Generating complex history...
====================
Iter | Modulo7 | Hist[ 4 ]==0 | XOR | Noise Flip | Final Outcome
------------------------------------------------------------------------
   5 |     0     |     1      |  1  |   False    |       1
   6 |     0     |     0      |  0  |   False    |       0
   7 |     1     |     1      |  0  |   False    |       0
   8 |     0     |     1      |  1  |   False    |       1
   9 |     0     |     1      |  1  |   False    |       1
  10 |     0     |     0      |  0  |   False    |       0
  11 |     0     |     1      |  1  |   False    |       1
  12 |     0     |     1      |  1  |   False    |       1
  13 |     0     |     0      |  0  |   False    |       0
  14 |     1     |     0      |  1  |   False    |       1
  15 |     0     |     1      |  1  |   False    |       1
  16 |     0     |     0      |  0  |   False    |       0
  17 |     0     |     0      |  0  |   False    |       0
  18 |     0     |     1      |  1  |   False    |       1
  19 |     0     |     0      |  0  |   False    |       0
------------------------------------------------------------------------
... History generation complete.

Creating training windows (size 16)...
Generated X shape: (984, 16, 1)
Generated y shape: (984,)
Data saved to branchnet_training_data.npz
```

Fig 3: Generating data for predictor training

5. Representation and Justification

While this simulation deviates significantly from the full complexity of ChampSim traces and the original BranchNet PyTorch implementation, it preserves the core functional differences between the predictor types:

- Traditional Predictors: The 1-bit, 2-bit, 2-Level, and Tournament implementations accurately reflect the fundamental state machines and history usage (local counters, global history correlation) of these standard hardware algorithms.
- ML Models:
    - Framework Choice: Keras/TensorFlow was used instead of the original PyTorch for ease of setup and implementation within this simplified context. The core concepts of layers, activation functions, loss, and optimization are transferable.
    - CNN: Represents the BranchNet idea of using convolutions to detect spatial patterns (though simplified here) within the recent history window.
    - LSTM: Represents adding recurrent layers capable of modeling longer-range temporal dependencies within the history window, conceptually similar to the CNN+LSTM model explored in [3].
    - Transformer: Represents exploring attention mechanisms, another modern sequence processing technique potentially applicable to identifying important historical branch correlations.
    - Simplification: These models lack the multi-slice geometric history, specialized embeddings (using simple windowed outcomes), and precise quantization/pruning of the original BranchNet, but they allow for a basic comparison of CNN vs. RNN vs. Transformer philosophies on the same sequential prediction task.

The synthesized dataset, with its XOR logic and specific historical dependency, was designed to create a scenario where simple local predictors would fail, global history correlation (2-Level) would offer some advantage, and ML models with sequence awareness (LSTM) or feature extraction (CNN) would have the potential to perform best by learning the specific underlying pattern.

6. Fusion Predictor Logic (Explored, Excluded from Final Simulation)

An attempt was made to implement a FusionPredictorMLAware that combined a basic predictor (e.g., 2-Level) with a trained ML model (e.g., BranchNet_LSTM). This involved:

- A selector mechanism similar to the Tournament predictor.
- Passing the history buffer to the fusion predictor's predict and update methods to allow the ML component to function.
- Careful handling of state updates (only updating the basic predictor's state within the fusion update).
- Adjusting accuracy tallying to account for ML model warm-up.

However, this introduced significant complexity into the simulation loop and resulted in excessive runtime due to frequent Keras model.predict() calls within the update logic needed for selector

evaluation. Good accuracy results (~100%) were also observed. But due to exhaustion of resources while running it with all the other predictors together, the fusion predictor was excluded from the final simulation (simulate_predictors_no_fusion was used) to focus on clear comparisons between standalone predictors.

7. Results Discussion and Representation

The simulation yielded the following accuracy results on the synthesized deterministic branch history (noise=0, Outcome = 1 if (i % 7 == 0) XOR (history[i-5] == 0) else 0):

PREDICTOR	ACCURACY
1-BIT	0.4840
2-BIT	0.4290
2-LEVEL	0.6480
TOURNAMENT	0.5900
BRANCHNET_CNN	0.8283
BRANCHNET_LSTM	0.9146
BRANCHNET_TRANSFORMER	0.5437

Table 1: Accuracies of various methods

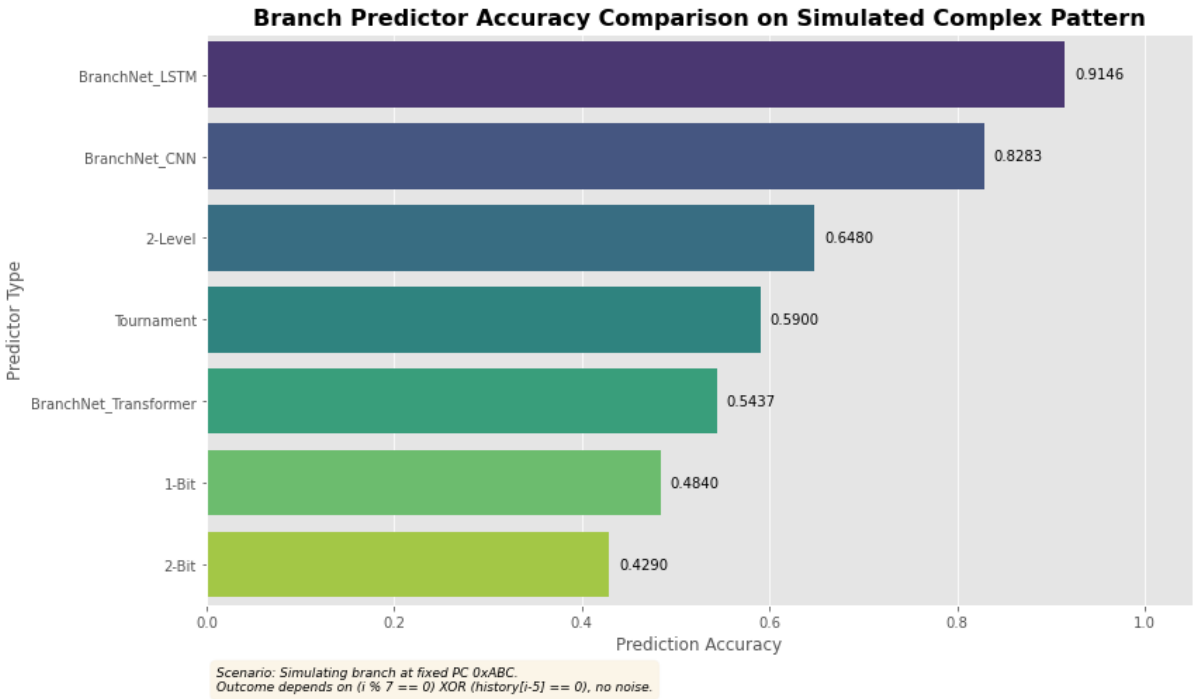

Fig 4: Comparison of accuracies

Analysis:

- Baseline Failure: The simple 1-Bit and 2-Bit predictors perform poorly (<50%), demonstrating their inability to capture either the periodic component or the specific historical dependency, especially combined with the XOR logic.

- Correlation Improvement: The 2-Level predictor shows a marked improvement (64.8%), indicating that correlating the fixed PC with the 4-bit GHR provides significant predictive information. It likely identifies recurring GHR states that statistically correlate well with the outcome, even without explicitly modeling the XOR or the i % 7 pattern.
- Tournament Mediocrity: The Tournament predictor underperforms the 2-Level predictor, suggesting that for this specific pattern, the simpler 2-Bit predictor (its local component) was often incorrect, causing the selector to potentially make suboptimal choices or take time to stabilize on favoring the 2-Level predictor.
- CNN Effectiveness: The BranchNet_CNN model significantly outperforms all traditional predictors (82.8%). This highlights the power of convolutional layers to extract relevant features (likely combinations of recent outcomes and potentially detecting the short-term patterns leading up to the history[i-5] state) from the history window, even without explicit recurrence.
- LSTM Superiority: The BranchNet_LSTM model achieves the highest accuracy (91.5%). Adding the LSTM layer allows the model to better learn the temporal dependency related to the state 5 steps ago, complementing the feature extraction performed by the initial CNN layers. This strongly supports the hypothesis that sequence-aware ML architectures are well-suited for branches with specific long-range historical correlations.
- Transformer Performance: The simple Transformer Encoder implementation did not perform as well as the CNN or LSTM models in this instance. This might be due to the model's simplicity, the relatively short history window (Transformers often benefit from longer sequences), or potentially requiring different hyperparameters or more training data to effectively utilize the attention mechanism for this specific pattern.

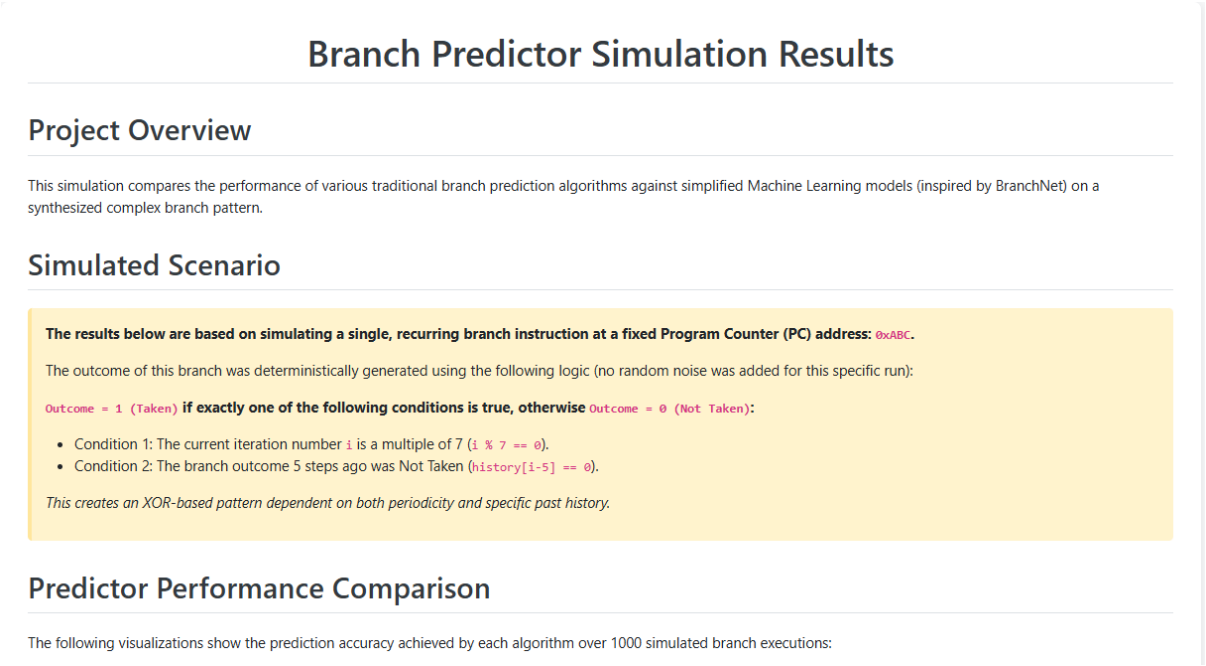

Fig 5: Results reported through a webpage

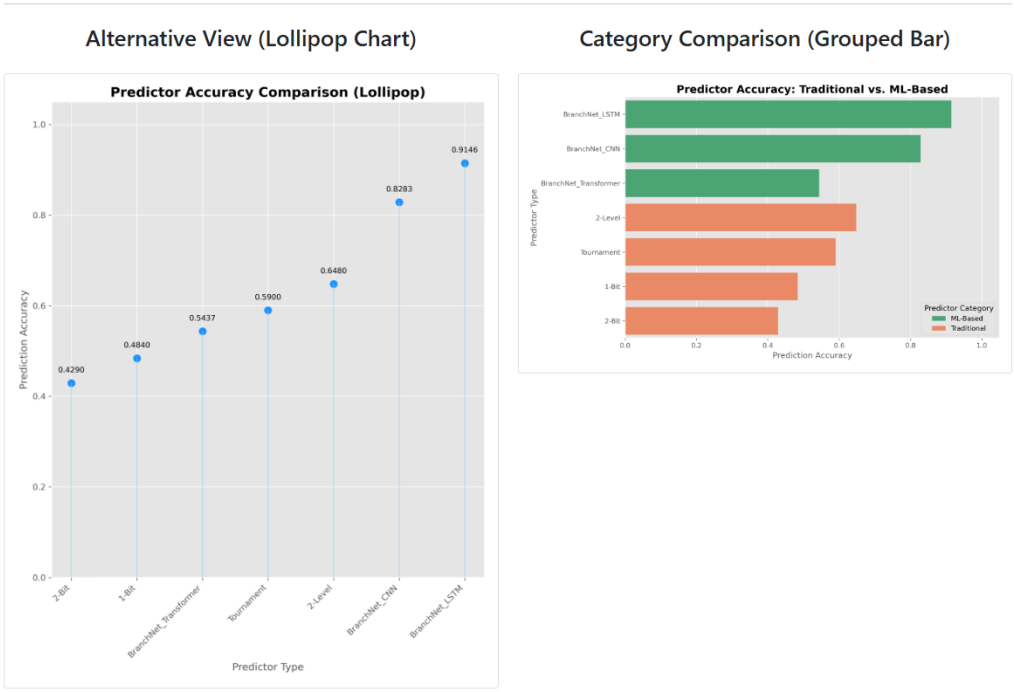

Fig 6: Results reported through a webpage – 2

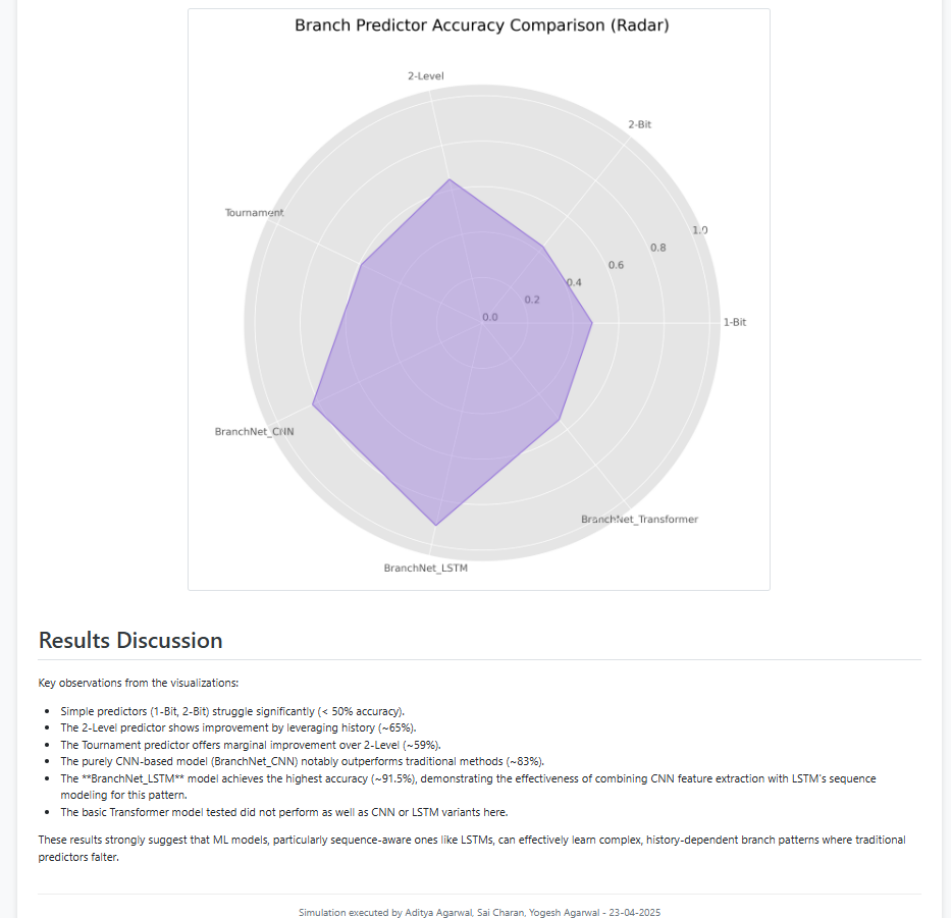

Fig 6: Results reported through a webpage - 3

Conclusion:

This project successfully demonstrated, within a simplified simulation environment, the potential for ML-inspired branch predictors to outperform traditional algorithms on challenging, history-dependent branch patterns. While direct replication of the complex BranchNet/ChampSim framework faced dependency issues, the developed Python simulation effectively highlighted the relative strengths and weaknesses of different approaches. The results clearly show that leveraging convolutional features (CNN) and particularly sequential modeling (LSTM) allows ML models to learn complex, non-linear correlations that fixed-logic hardware predictors struggle with, achieving significantly higher prediction accuracy on the synthesized benchmark. This motivates further research into practical hardware implementations of such ML-based prediction techniques.

8. References

[1] Quangmire/ChampSim: ChampSim repository. (URL: e.g., https://github.com/ChampSim/ChampSim)

[2] Tarsa, S. J., Lin, C. K., Keskin, G., Chinya, G., & Wang, H. (2019). Improving Branch Prediction By Modeling Global History with Convolutional Neural Networks. arXiv preprint arXiv:1909.03891. (Implicitly referenced by thesis)

[3] Vontzalidis, A. (2021). Branch Prediction using artificial neural networks [Diploma Thesis]. National Technical University of Athens. GitHub: https://github.com/aristotelis96/thesis-branch-prediciton-ml

[4] Zouzias, A., Kalaitzidis, K., & Grot, B. (2021). Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies. ML for Computer Architecture and Systems 2021. arXiv:2106.13429.

[5] SPEC CPU 2017 Benchmark Suite. https://www.spec.org/cpu2017/

[6] Paszke, A., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in Neural Information Processing Systems 32.

[7] S. Zangeneh, S. Pruett, S. Lym and Y. N. Patt, "BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches," 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 2020, pp. 118-130, doi: 10.1109/MICRO50266.2020.00022.

[8] BranchNet source code: https://github.com/siavashzk/BranchNet