Building a Lightweight Branch Predicting Transformer

Cameron Crow (cc3385) & Ryder Walsh (rw7049)

ECE475 Term Project

6 May 2025

Abstract

Branch prediction has proven to be a central optimization problem in computer architecture. Here, we implemented a novel branch predictor as a tractably hardware-implementable transformer, in response to the impressive performance of OpenAI's GPT models. Sequences of assembly instructions operate similarly to sequences of words, with "grammatical" conventions and temporally causal relationships. By giving the transformer a sequence of instructions (always ending in a branch) and the state of the register file at the beginning of the sequence, we expect that it will be able to learn whether the terminating branch will be taken or not. We have collected prediction accuracy data under various training and test conditions and have conducted ablation studies to try to maximize the performance-to-memory size ratio, reaching prediction accuracies as high as **95.2%** in some cases. Overall, it is unclear whether the prediction accuracy justifies the additional hardware cost and computational overhead. However, we do briefly explore some promising benefits of this inference paradigm that emerge, as well as theoretical areas of optimization that can be further explored in future studies, such as KV caching opportunities, preemptive loop unrolling, and more conservative bit resolution, indicating some promise in the area of branch prediction with deep learning architectures.

Objective: Branch Predicting Transformer

The driving question of our project is the following: Can we create a lightweight transformer model that accurately predicts branches? "Lightweight" in this context encapsulates the "cost" of the transformer—the amount of memory required to implement it, any additional arithmetic units, and the overall complexity it adds to a microprocessor. Our model resembles a large language model, designed to predict the outcome of control flow instructions. Given a sequence of disassembled x86 instructions and the respective register file states leading up to a branch instruction, the model predicts whether the terminating branch will be taken. To generate training data, we used Intel's PIN dynamic binary instrumentation tool to trace program executions on Princeton University's Adroit computer cluster—a Linux machine. Each line of data in the output of the PIN execution included the disassembled instruction sequence, register values, and labels for the true branch outcome. The prediction model was implemented in PyTorch, allowing us to utilize its broad suite of machine learning tools, including its built-in support for backpropagation and training loop management. Our research aims to frame branch prediction as a machine learning problem, where patterns in instruction execution and data inform speculative execution.

Methodology

Transformer Paradigm

Our transformer architecture follows the following general architectural structure seen in Figure 1. The input data gets passed through an embedding process—see Figure 2—that the transformer has learned in order to represent the information in useful, interpretable vectors. This embedding then gets passed through several layers of basic self-attention transformer blocks. At the output, the matrix gets projected into a 1x2 vector that holds the prediction for the branch outcome. The two values in the vector sum to 1, each representing the probability of a branch or fall through.

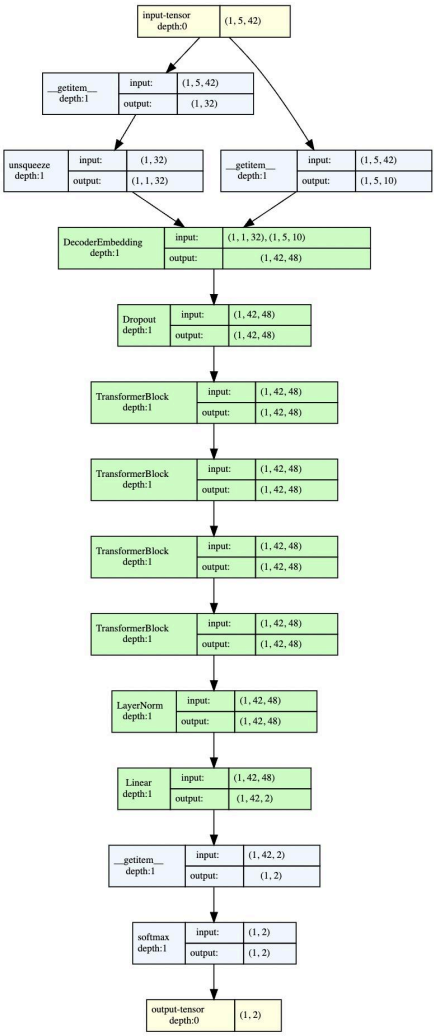


Figure 1: Flowchart representation of the architecture of our branch predicting transformer.

Data Representation

The transformer receives instruction and register file information from the processor. The inference paradigm is as follows:

1. The sequence of instructions from the current instruction all the way to the next branch are passed to the transformer, as well as the current state of the register file. This information flows through the transformer to make its prediction.
2. At the next cycle, the new sequence of instructions and new state of the register file are passed to the transformer. This new sequence is almost identical, though it lacks the first instruction from the previous pass, as that has already been issued. The register file has been updated by instructions writing to it in the cycle that has passed.
3. This continues until the branch has been reached, at which point the most recent prediction determines which instruction is fetched next.
4. Repeat this sequence until the end of the program.

The exact instruction and register file information that the transformer receives is described in more detail in the next section, *Gathering Training Data*.

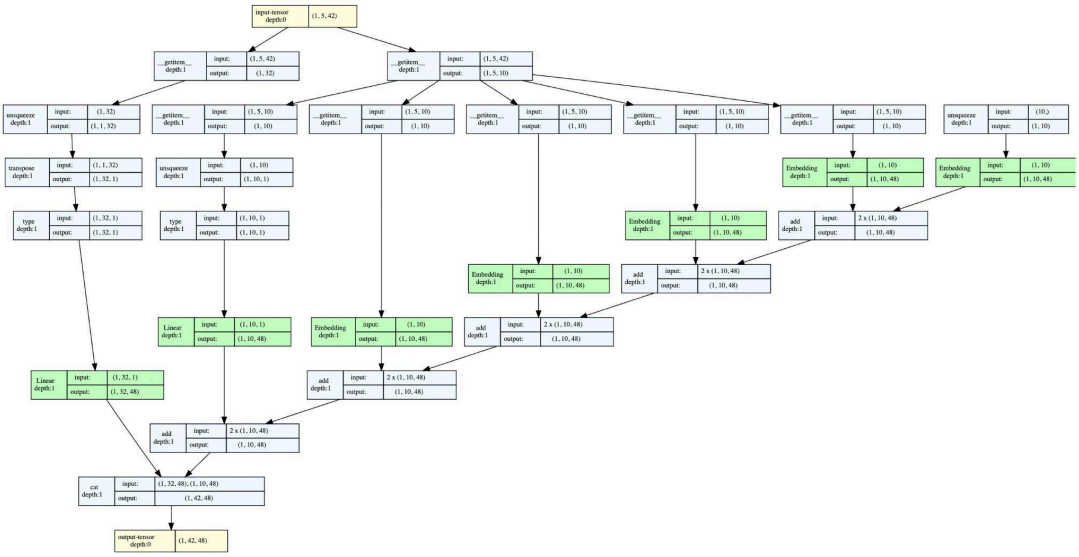


Figure 2: Flowchart representation of the embedding process used to translate the training data into interpretable vectors.

Gathering Training Data

Since we were training the transformer via supervised learning, our training process required large amounts of data—i.e., many instruction-label pairs. Hand-tracing programs and labeling instructions as branches and taken/not taken would render the project intractable. Instead, we utilized a dynamic binary instrumentation tool called Intel PIN. Coded in C++, PIN allows tracing a program at runtime to gather data about its execution.

We wrote a PIN program that, for every executed instruction in a program, outputs the program counter (instruction cache address; PC), the string representation of that instruction, two boolean values representing if the instruction is a branch and if it is taken, a list of the operands (immediate and source registers), and the registers set by that instruction (destination, stack pointer, and flag registers).

```
15303aecf59f | jz 0x15303aecf690 | 1 0 | {no imm} | rsp rdi  | {no dest}
```

Figure 1: A sample line from the instruction parser when run on *ls*. We can see the fields described above. Note that this single line provides the information that the branch was not taken, and the registers RSP and RDI were relevant to this determination and/or execution of the instruction.

Simultaneously, in a separate output file, the state of the 16 x86 general-purpose registers is dumped. These general-purpose registers are specified in the “Intel 64 and IA-32 Architecture Software Developer’s Manual Volume 1: Basic Architecture” (Intel, 3-13, March 2025).

```
0x1530269845b5 0x2f92593f00 0x7ffda808ae28
0x7ffda808acb0 0x153026985570 0x000000063
0x1530269845ba 0x310eec095a 0x15302768ea88
0x000000001 0x15302768ea88 0x153027b14af0
0x15302768ea88 0x1530269811a0 0x153027b15a80
0x1530269845b5
```

Figure 2: A sample line from the register file dump output file for “ls”. No newline characters were inserted within a single dump instance to minimize output file size.

Running our program on the built-in “ls” executable on Adroit yielded over 1.2 million executed instructions and thus 2.4 million lines of text among our two outputs and approximately 400MB of data. Repeating this process for a suite of Linux built-in binaries proved efficient and fruitful. This suite includes *cat*, *date, echo, env, head, hostname, ls, pwd, uptime*, etc. Gathering data on these executables alone provided us with 3GB of data with which to train a 1MB transformer. These data were then parsed and translated into tensor format for use in our supervised training loop.

Testing

We chose to incrementally test our model in order to test its ability to learn with varying training set sizes and generalize. From our suite of built-in Linux binaries, we designated *env* as our benchmark test due to its balance of repetitive string processing and directory tree navigation.

We first trained our 1MB model on a subsection of 40,000 data points from the 1.2 million training data points generated from *ls*. We then tested the model on that same training set to ensure that it was learning properly. To test the model’s generalization, we also tested it on *cat*, *date*, *head*, *pwd*, and *env* before continuing training.

Next, we compiled a training set by choosing subsections of the data from all traced programs except for *env*. This comprehensive dataset diversified the training of the model in hopes of improving overall accuracy and generalizability. To reduce training time and improve tractability as a hardware implementation, we reduced the size of the model to 933,376 parameters. We then tested the model on *env*, our benchmark dataset, where we would be most critical of accuracy.

Finally, we repeated this comprehensive training and benchmark testing process for models of varying parameters: number of transformer layers, attention heads, and hidden dimensions.

Discussion

Our branch predictor follows a unique prediction paradigm that differentiates itself from mainstream predictors like TAGE or BHT. More conventional predictors invoke their predictor

logic in/right before the fetch stage. This is permissible, since the history table search logic is largely computed in parallel, so the prediction latency is minimal. Conversely, our transformer is largely a series computation, since information must get passed sequentially from one layer to the next. Thus, latency in our model is much greater. Luckily, the paradigm aligned nicely with some emergent qualities that help this issue:

1. Our inference paradigm means we can predict a branch as far as context_length instructions in the future, which means the predictor could be pipelined with context_length stages without any penalty. In our transformer, context_length = 20. Increasing context_length leads to greater overall size/latency but allows for prediction farther in advance.
2. As instructions pass and the instruction pointer approaches the branch, fewer computations are required as the distance to the branch decreases (further discussion with KV caching).
3. Predictions closer to the branch are generally more confident; predictions farther from the branch are generally more speculative. A more comprehensive prediction might incorporate this fact, weighting more recent predictions heavily to determine its final prediction before fetching.
4. If the predictor is pipelined, then some predictions will be computed after the first speculative instructions have already been fetched. In the event that the prediction flips at this point, the instruction pointer can adjust to the newest prediction without much penalty.

These are 4 qualities that naturally emerged as a result of the inference strategy. There are also several potential areas for optimization which could be further studied:

1. KV caching is an inference technique used by modern transformers, where the model stores intermediate information that will remain unchanged for the duration of the inference, mitigating unnecessary recomputation. Though our inference is slightly different from OpenAI's GPT models (their context length grows as its token predictions are appended to the sequence, while ours shrinks as we complete instructions and no longer need them for prediction), we still have information we can reuse. The first prediction in a sequence computes the keys and values for all instructions, all of which can be reused for successive predictions as the program approaches the branch.
2. Once the prediction passes some chosen confidence threshold (~95%), the predictor could finalize that prediction and expand its horizon to the next branch, even if the closest one has not yet been reached. This unrolling allows the predictor to perform against branches in close succession.
3. By reducing the bit resolution of the parameters, we can significantly save on static memory budget. Since the task is binary classification (branch taken or not), it is conceivable that reducing the precision might not affect the final prediction, since the resolution of the output vector is already very low precision (0 or 1). Storing parameters as bytes means the total number of parameters equals the total number of bytes.

Results

We first trained on a small subsection of the *ls* program, and wanted to see how prediction ability generalized to other Linux boot programs. Testing on the training set (which is technically poor practice in machine learning) yielded the best results, not surprisingly. Other Linux boot programs performed well, but not stellar.

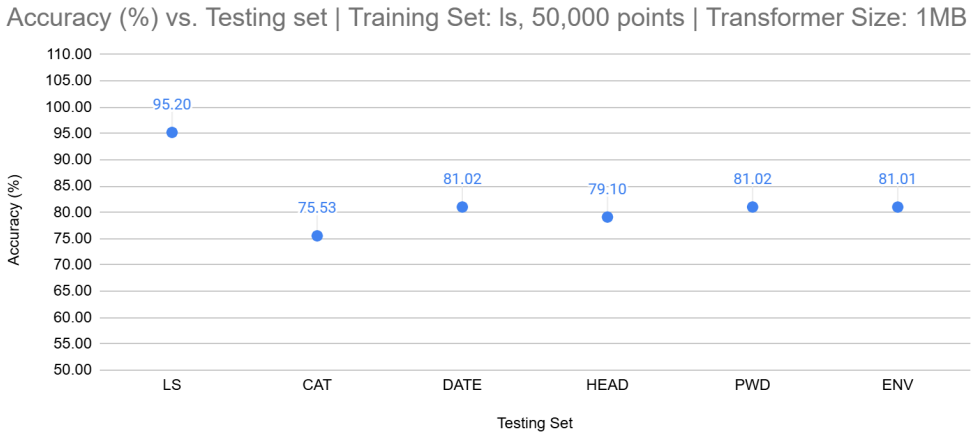
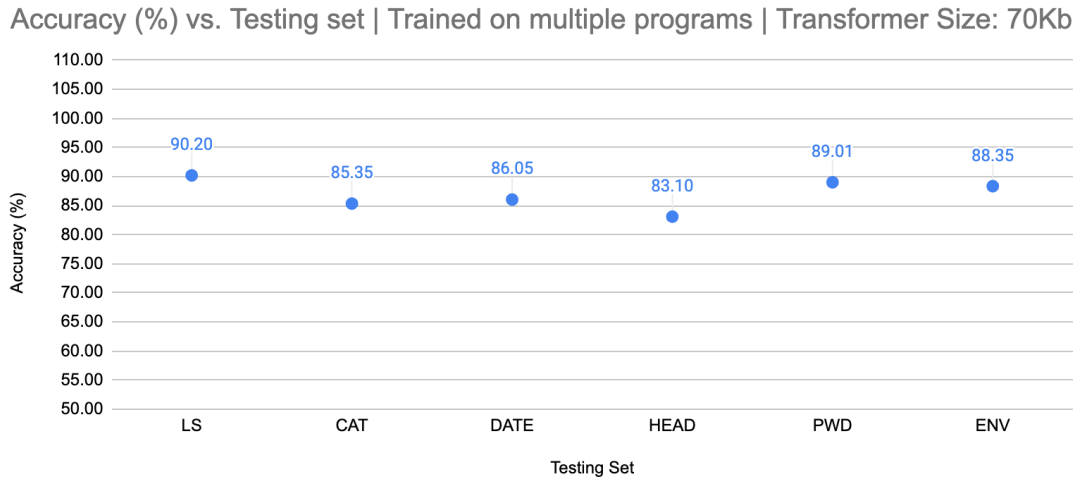


Figure 3:

We then wanted to train on a more diverse selection of programs, which should boost generalization ability of the transformer, leading to higher performance on more programs. We trained on a greater selection of Linux boot programs to observe this. We also wanted to conduct ablation studies to see how the size of the transformer affects predictive ability. Recall that we aimed for a lightweight transformer. A larger transformer trained on more data with more computing resources would be almost guaranteed to perform better, but would be intractable as a hardware implementation, and would thus lack insight. We did these two simultaneously due to limited computing resources and time constraints. Performance hovers around 86%.



Related Work

Augmenting Dynamic Branch Predictors with Static Transformer Guided Clustering, Avaneesh Deleep

Research conducted by Avaneesh Deleep at Imperial College London combines our static transformer with traditional, dynamic branch predictors. In our project, we pre-train the model in order to avoid the massive computational cost of compile-time or runtime training. Our model also makes predictions on a per-instruction basis. Deleep's "hybrid" model, however, uses a transformer to categorize instructions via "color labels". These color labels are fed into a custom branch predictor that more closely resembles traditional hardware branch predictors. By pre-filtering the instruction data this way, the branch predictor is able to make more accurate predictions than some state-of-the-art predictors (Deleep, 2024).

Dynamic Branch Prediction with Perceptrons, Daniel A. Jiménez, Calvin Lin

This research, conducted by Daniel A. Jiménez and Calvin Lin at the University of Texas at Austin, focuses on a very simple neural network: a single layer of perceptrons. The project maintains a much lower memory budget than our transformer—4KB, as opposed to our maximum budget of 1MB. What sets our model apart from their work is its higher dimensionality, which contributes to its increased memory budget. Furthermore, their model is dynamically trained. After the true branch outcome is computed, the weights contained by a perceptron are updated, thus increasing accuracy as time goes on. This is enabled by the small amount of information associated with the perceptron making the prediction. This is unlike our pre-trained model, which requires a massive amount of vector computations to train (Jiménez, Lin, n.d.).

References

Deleep, A. (2024). *Augmenting Dynamic Branch Predictors with Static Transformer Guided Clustering* (thesis). Imperial College London, London, UK.

Intel. (2025, March 5). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Jimenez, D. A., & Lin, C. (n.d.). *Dynamic Branch Prediction with Perceptrons* (thesis). The University of Texas at Austin, Austin, TX.