A Study of Branch Prediction Strategies

James Smith

Presented by:
Yuhan Cai
Ning Chen

Overview

- Branches break smooth flow of instruction fetching and execution.
- Predict branch direction to reduce delay.
- A wrong prediction may lead to more delay.
- Goal: maximize the likelihood of correct predictions
- 6 programs are used in this study
  - Chosen for their unpredictability
- 7 prediction strategies

Preliminary Categories

- Static prediction
  - Does not use past history
  - Success rate vary widely & Program sensitive
- Dynamic prediction
  - Varies based on past history
  - Less program sensitive
  - Use one or more recent executions
  - Use memory to keep a history table

Strategy #1

- Prediction: all branches will be taken
- Widely different accuracies
- Static strategy
- Simple and cheap
- Strategy 1a: predict all branches with certain codes will be taken
  - Somewhat tuned
  - Greater accuracy

Strategy #2

- Prediction: a branch is decided the same way as it was last time.
- Provides better accuracy
- Dynamic strategy
- Less program sensitive
- NOT physically realizable

Strategy #3

- Prediction: all backward branches are taken
- Baseline: loops are terminated with backward branches
- Often works well, sometimes better than #2
- High program sensitivity
- Needs to do computation prior to prediction

Strategy #4

- Table: most recently used branch instructions NOT taken.
- Predict a branch instruction will not be taken iff it's in the table. Use LRU replacement to add new entries if necessary.
- Bigger table → better performance

Strategy #5

- One bit for each instruction in the cache, used to record if it was taken last time
- Branches are predicted to be decided as on their last execution
- If a branch has not been executed, it's predicted to be taken
- Results close to #2

Strategy #6

- Hash branch instruction address to *m* bits
- Use the index to address a RAM containing the outcome of the most recent branch instruction indexing the same location
- Prediction: branch outcome is the same
- Performance similar to #2

Strategy #7

- Use a counter instead of a single bit
- Prediction: taken iff positive
- Increment counter if a branch is taken, decrement it otherwise.
- Usually better than previous strategies
- A natural way to hiearchical prediction
- Predictions made at extremes are more accurate

Analysis of Correlation and Predictability: What makes Two-Level Branch Predictors work

M. Evers, S.J. Patel, R.S. Chappell, Y.N. Patt

Motivation

- Design of branch predictors
- Studies on which predictors and configurations best predict the branches in a given set of benchmarks

- Little research has been done on which characteristics of branch behavior make predictors perform well

Correlations

- Direction correlation

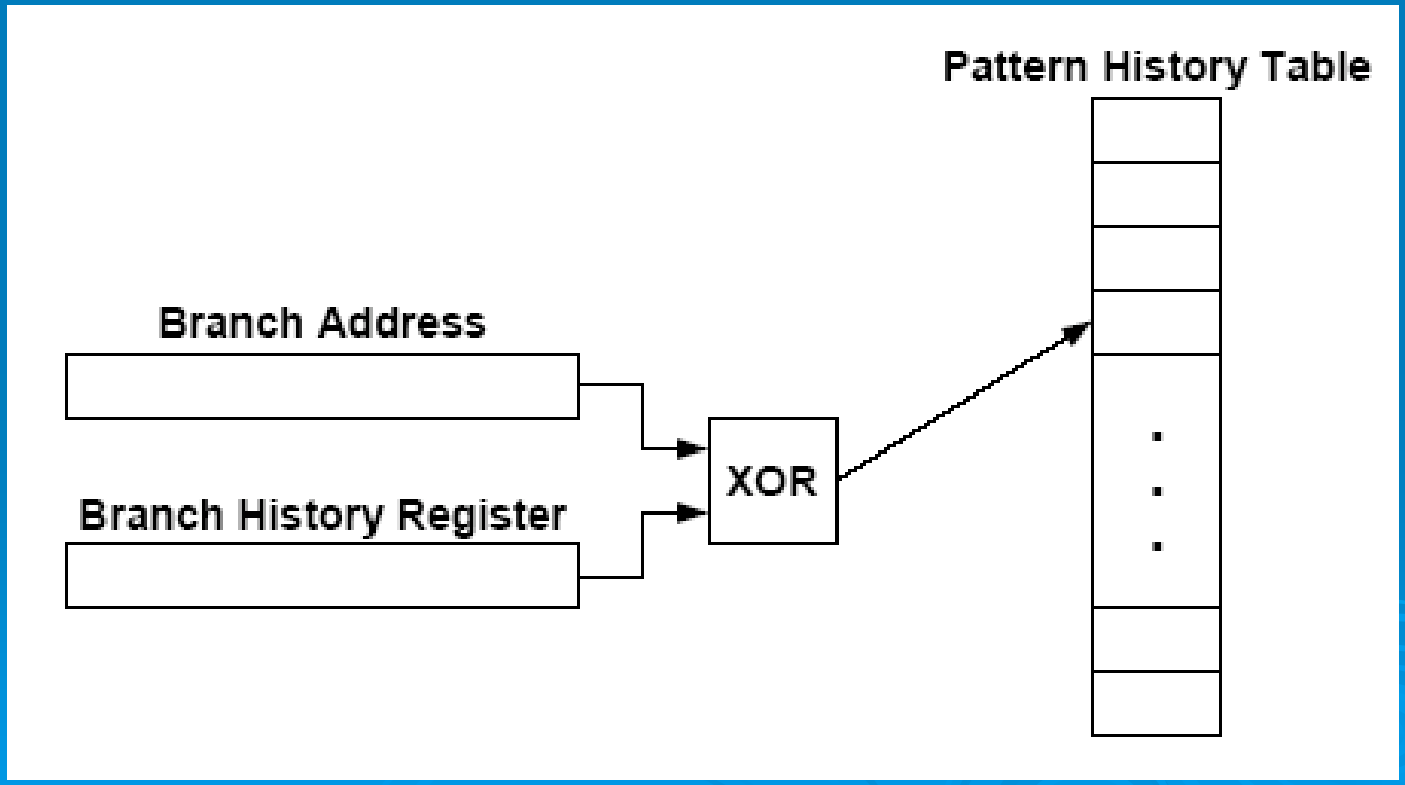
branch Y: if (cond1) ... branch X: if (cond1 AND cond2)	branch Y: if (cond1) a = 2; ... branch X: if (a == 0)
(a)	**(b)**

Correlations (cont.)

- In-path correlation

```
branch Y: if (NOT(cond1)) ...
branch Z: else if (NOT(cond2)) ...
branch V: else if (cond3) ...
...
branch X: if (cond1 AND cond2)
```

Correlation in two-level branch predictors



Experimental results

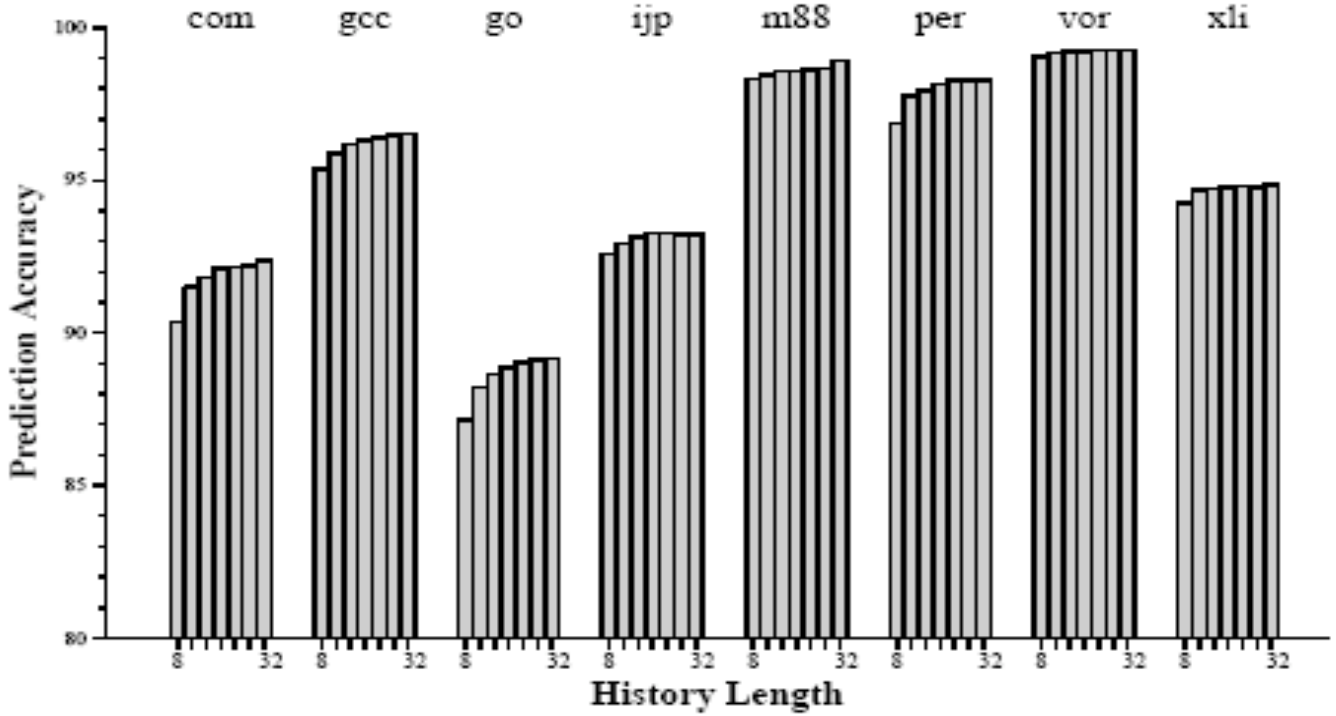
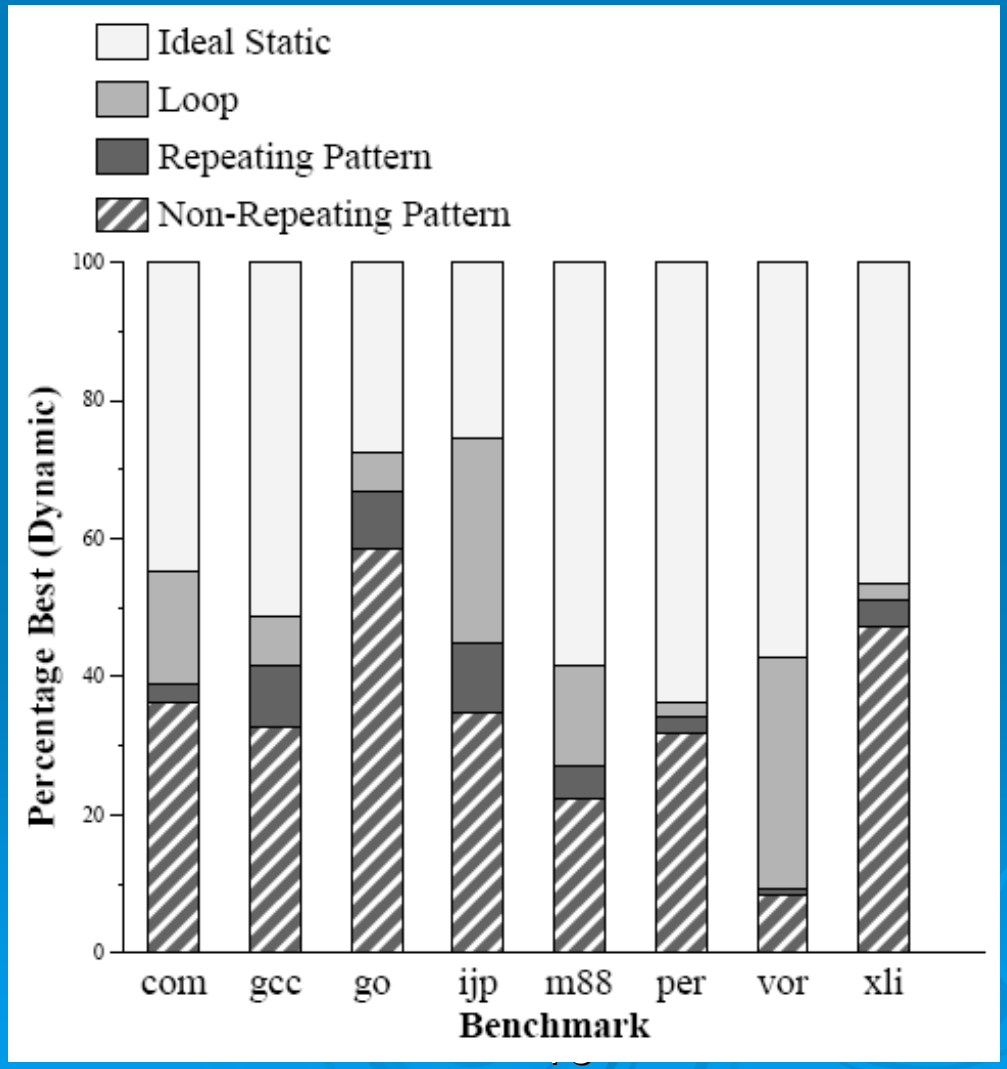


Figure 5. Accuracy as a function of history length using a 3-branch selective history

Per-address predictability

- Loop-type branches
  - for-type and while-type
- Branches having repeating patterns
  - fixed-length patterns and block patterns
- Branches having non-repeating patterns

Distribution of per-address predictability classes



Experimental results

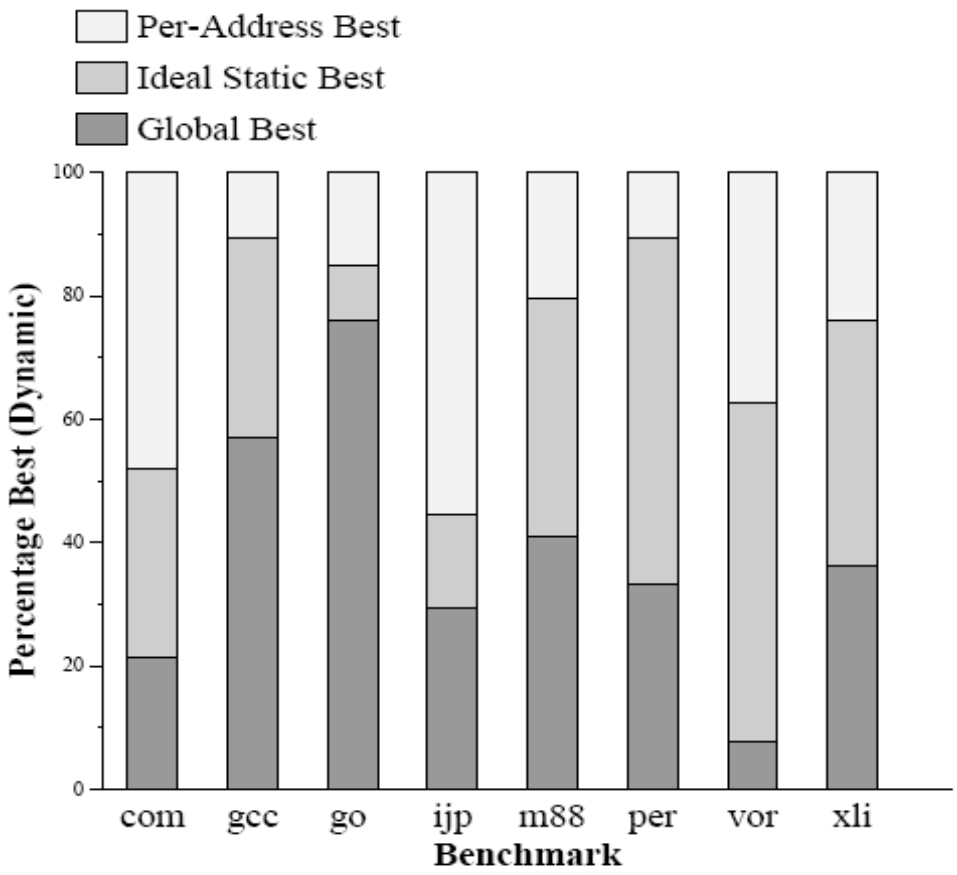


Figure 8. Distribution of branches best predicted using global correlation, per-address based predictors, and an ideal static predictor, weighted by execution frequency

Discussion

- What is the cost of a hybrid branch predictor? Although it achieves better performance, it cost more than global predictors and per-address predictors?
- Would there be an advantage to allowing compilers to flag certain branches as to how likely they are to be taken using some higher-level knowledge they have? Is it reasonable to think that compilers could even do that with an acceptable accuracy level?
- Would actual machine learning techniques be better at making branch predictions? Could the models be made simple enough to be implemented efficiently in hardware?
- Are machine learning techniques used inside any real processors?
- a global predictor is the same thing as a correlation-based predictor, right?
- The number of branch history patterns increases exponentially with the number of branch instructions of the history?
- How can we account for loop types when *n* is not known or dynamically changes?
- Easy to see which one *would have* been better after branch is executed. But, how can we know which prediction in best *a priori*? (This is the big hole in this paper!)