1 Description of Modifications

Only file `sim-safe.c` was modified and all predictors are implemented in this file. When I implemented predictor *v*, the program used more memory than permitted. Thus predictors *i-iv* can be run at the same time, but predictor *v* must be run separately. Switch modes by changing the define on line #349 : `#define RUN_I_THROUGH_IV true`. All changes are in the `sim_main` function, with the exception of the misprediction counters for each predictor, globally defined arrays, and define statements for array dimensions.

Predictor *i* gets the index value, looks up the last outcome from the `bpred_pht_i` array as the prediction. Then it replaces the spot in the array with the the new outcome.

Predictor *ii* gets the state (0-3) from the `bpred_pht_ii` array. If $state \geq 2$, the branch is predicted taken. It increments the state if the outcome is taken and decrements it if not taken (max 4, min 0).

Predictor *iii* uses a 2D array. It gets its prediction by indexing `bpred_pht_iii` with the index and with `last_outcome_iii`, which is a value from 0-1. Then it updates the predicted outcome to be the actual outcome, and sets the `last_outcome_iii` variable to be the outcome of this iteration for the next cycle.

Predictor *iv* is also a 2d array. However instead of the second index being a value between 0-1, it is a value between 0-15. These 16 instances represent all the possible sequences of history: 0000 (all not taken), 0001 (all but most recent not-taken), etc. Each of these gets a saturating counter that works the same as in predictor *ii*. To update the history, we shift all bits left to create a spot for our new history entry. We then AND the history with 15 to only retain the last 4 bits. Then if the outcome was taken, we set the most recently added bit to be 1.

Predictor *v* works in exactly the same way as predictor *iv* except we retain more history and have less entries in the table. Instead of AND-ing our `branch_history_v` with 15 to preserve the last 4 bits of history, we and it with $2^{HISTORY\_TO\_RETAIN}$ to only keep last `HISTORY_TO_RETAIN` bits of history. In my final submission, the history to retain was 18 instances, corresponding to a value from 0 to 262144. This exceeded the allowable size of an array, so I reduced the number of instances to 512.

2 Interpretation

The trend of prediction accuracy of the different predictors was consistent between all benchmarks. The accuracy of predictors in order from best to worst is:

1. 2-bit saturating counter predictor using 4-bits of branch history
2. 2-bit predictor with no branch history
3. 1-bit predictor with 1-bit of branch history
4. 1-bit predictor

When limited to two bits, implementing extra states to store "weakly" taken/not-taken versus "strongly" taken/not-taken information takes priority over branch history. Programs are heavily iterative and the 1-bit predictor will still suffer from 2 mispredictions every loop.

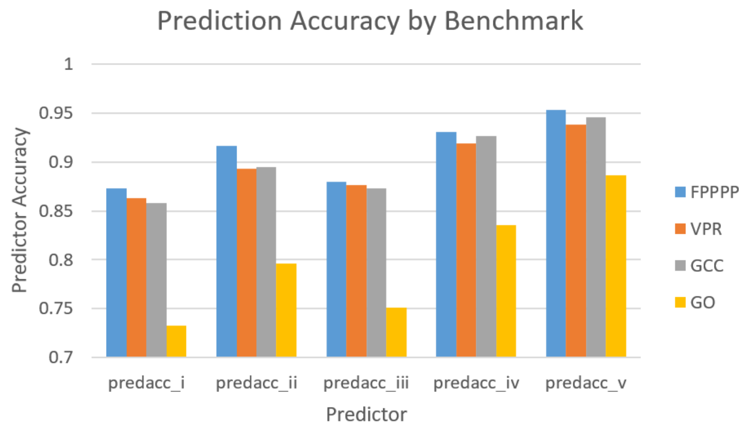


Figure 1: Prediction Accuracy of each Benchmark by Predictor

fpppp has the best performance as FP programs have higher accuracy prediction. FP programs have a higher percentage of taken conditional branches because they have longer looping structures and less if-then-else statements. This higher taken frequency corresponds to higher prediction accuracy (Cheng, 2000).

All predictors have worse performance for the *go* benchmark. This is because a) *go* is solely an integer program, and b) *go* has "many small loops and lots of control flows" (Cheng, 2000), thus branch predictors designed for long loops will perform worse.

In predictor *v*, I prioritized devoting resources to branch history rather than entries in the `bpred_pht_v` array. In my final version, I only had 512 entries in the branch predictor, but retained 18 instances of branch history. Beyond 18 states, reducing the number of entries further caused the prediction accuracy began to suffer slightly, as seen in Figure 2.

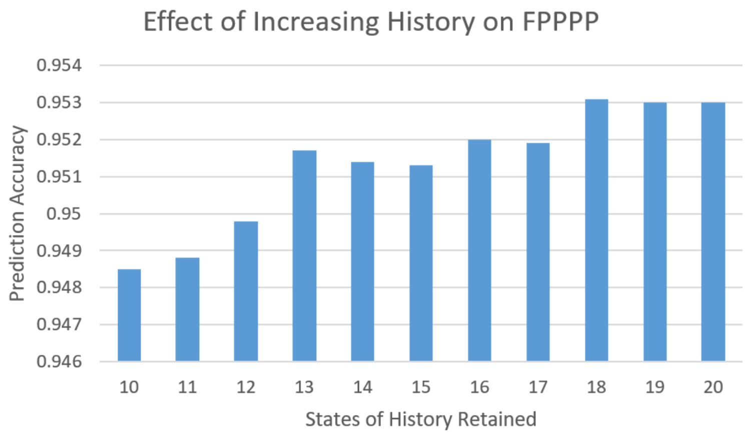
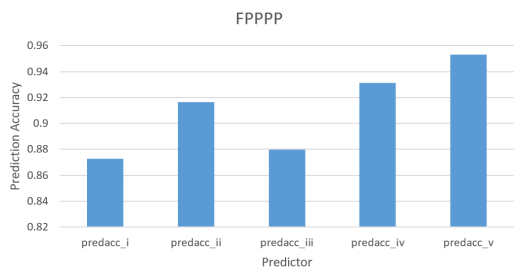


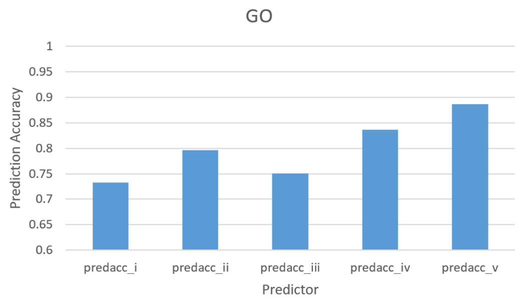
Figure 2: Prediction Accuracy of FPPPP with More Resources Devoted to History

In lecture we learned that a table with 4000 entries has comparable performance to an infinitely sized table. My tests reaffirms that extra entries have diminishing returns. The eventual decrease in accuracy with history may also be influenced by the results of unrelated branches. With larger history, the predictor is more vulnerable to irrelevant noise from uncorrelated branches. This noise means the predictor takes longer to train specific branches (Evers, 2000).

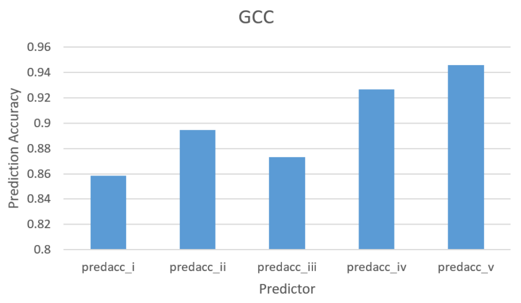
3 Appendix



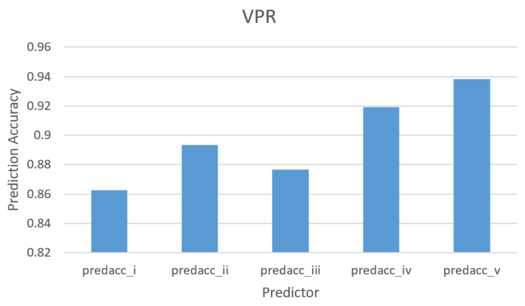
(a) fpppp



(b) go



(c) gcc



(d) vpr

Figure 3: Percentage of Executed Instruction Types by Benchmark

Benchmark	**Predictor$_i$**	**Predictor$_{ii}$**	**Predictor$_{iii}$**	**Predictor$_{iv}$**	**Predictor$_v$**
Go	0.7329	0.7963	0.7509	0.8359	0.8865
GCC	0.8584	0.8945	0.8732	0.9267	0.9460
VPR	0.8627	0.8933	0.8765	0.9193	0.9384
FPPPP	0.8728	0.9164	0.8796	0.9311	0.9531
Mean	**0.8317**	**0.8751**	**0.8450**	**0.9032**	**0.9310**

Table 1: Average Accuracy of Different Predictors

References

Cheng, C.-C. (2000). *The schemes and performances of dynamic branch predictors.* Technical Report, Berkeley Wireless Research Center.

Evers, M. (2000). *Improving branch prediction by understanding branch behavior* (Unpublished doctoral dissertation). University of Michigan.