6.375 Complex Digital Systems
Francis Wang, Shana Mathew

Multi-tiered branch predictor for pipelined processors

Background

In processor design, branch predictors are hardware structures that make a prediction on the direction of branches before the outcome is known for certain. This prevents the processor from stalling when faced with control hazards. Instructions after the branch are fetched speculatively. If the prediction is found to be correct, the processor proceeds without incurring any additional delays. However, if the prediction is found to be incorrect, the pipeline is flushed and the fetch unit is redirected to the correct address. For modern out-of-order superscalar processors, the fetch stage is decoupled from the execution units and there is a large delay between when a branch instruction is fetched and when the branch is resolved. This makes branch mispredictions very costly and motivates the development of highly accurate branch predictors.

Some modern processors employ a tiered strategy to branch prediction. There is a hierarchy of prediction structures where each predictor is slower, larger, and more accurate than the last. Closest to the fetch stage is the Branch Target Buffer (BTB). The BTB is very tightly coupled to the fetch stage and it predicts the next address before the instruction is even fetched from memory. When an instruction is identified as a branch, the first level branch predictor kicks in and redirects the instruction flow if it believes the BTB to be in error. As the instruction goes through the pipeline, more sophisticated branch predictors are consulted and the instruction flow is redirected as is appropriate. These branch predictors may take several cycles to compute their results but make use of longer history records or are specialized to recognize certain program behavior such as loops or function returns. The combination of small and fast predictors and large and slow predictors allow designers to strike a good compromise between latency and accuracy.

Overview

In this project we implement a three-tiered branch predictor structure for a pipelined R32I RISC-V processor consisting of a BTB and two branch predictors of varying sophistication. Later predictors can override the decision made by earlier predictors by flushing the pipeline and redirecting the fetch stage. There are two major components to this project, the handling of speculative state in the pipeline and the implementation of the predictors. For the pipeline aspect of this project, we have to ensure that all right path instructions are executed and all wrong path instructions are squashed. We test the correctness of the pipeline by running assembly and C benchmarks. The predictors themselves are verified by comparing trace dumps of the processor with software models to check that all of the predictions match up with the models. Finally, we present some results on the effect of table size on predictor accuracy.

High-level Design

We leverage the infrastructure of the processor lab in our project by modifying the three stage bypass processor. The multi-tiered predictor structure that we propose is only suitable for deeply pipelined processors. We emulate such a processor by splitting the decode stage from the execute stage and inserting two additional decode stages. The simple structure of the pipeline allows us to abstract away features of a more complex processor such as superscalar fetch or out-of-order execution and focus on the branch prediction aspects of the design. The two specific branch predictors that we implement are the Gshare and TAGE predictors. The Gshare predictor is a simple and effective branch predictor that indexes an array of two-bit saturation counters with a hash of the branch address and the global branch history [1]. Its straightforward structure makes it a good candidate for a small and fast predictor that is tightly coupled to the fetch stage. In our implementation, the Gshare predictor returns its result within a single cycle. The TAGE predictor is a more sophisticated predictor that makes use of multiple history lengths and an array of partially tagged tables to capture correlations from both remote and recent branch history [2]. It is one of the highest performing branch predictors for which the details are publicly known. Its high accuracy and more complicated prediction generation logic makes it a suitable candidate for our secondary predictor. In our implementation, the TAGE predictor returns its result in the next clock cycle. A description of the pipeline stages of our processor is presented in Table 1. Stages that may trigger a redirection are marked with an asterix.

FE	Instruction requested from memory, BTB predicts next PC
D1*	Instruction received from memory, simple control flow instructions resolved, prediction received from Gshare
D2*	Prediction received from TAGE
D3	Data fetched from register file, destination register added to scoreboard
EX*	Instruction executed, branches resolved, predictors updated, loads and stores launched
WB	Loads received from memory, destination register written to and removed from scoreboard

Table 1. Description of pipeline stages

Speculative state is managed by a set of three boolean epoch counters, corresponding to the three stages of the pipeline where a redirection could occur. Each instruction carries an epoch, and if the epoch does not match that of the epoch counters the instruction is dropped. When the instruction stream is redirected, the epoch counters are updated, invalidating all of the younger instructions in the pipeline. When multiple redirections occur in the same cycle, the redirection furthest down the pipeline takes precedence. In the execute stage, branches are resolved and the predictors are updated according to their individual update policies. A diagram of the processor pipeline is presented in Figure 1. Predictors are aligned vertically with the pipeline stage in which it takes effect.

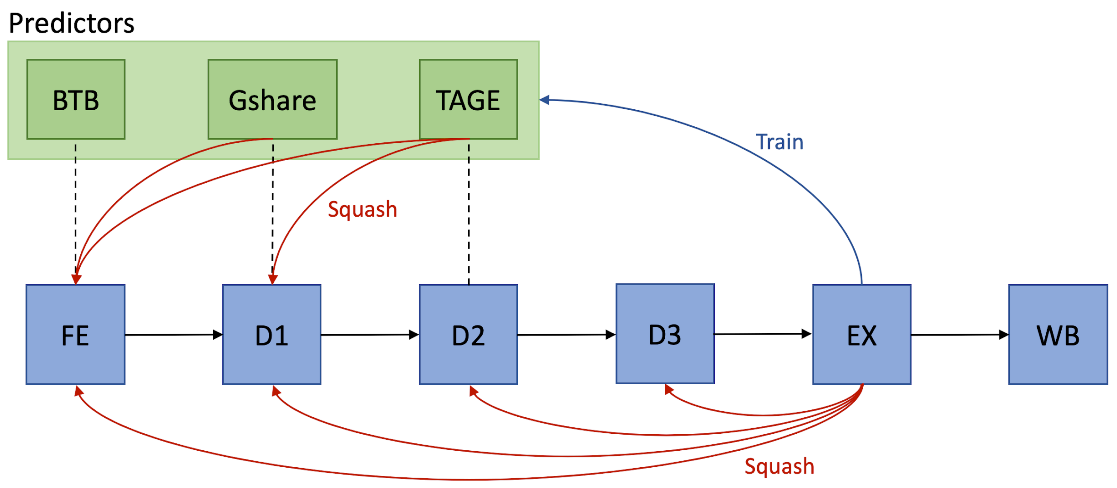


Figure 1. Pipeline diagram with predictors

Now we will discuss the performance benefits of the multi-tiered prediction structure. Suppose that the prediction generation logic for the TAGE predictor cannot be evaluated in a single cycle. If we have only the BTB and the TAGE predictor, the prediction penalties for the different cases are shown in Table 1. The average misprediction penalty is given by $N_{BT} = 4 - 2P_T - 2P_TP_B$.

BTB	**TAGE**	**Penalty**
X	0	4
0	1	2
1	1	0

Table 2. Penalty with BTB and TAGE predictor

However, if we insert the Gshare predictor, which is capable of making a prediction combinationally within a single cycle, we can potentially redirect an incorrect branch one cycle earlier. The new prediction penalties are shown in Table 2 and the average misprediction penalty is given by $N_{BGT} = 4 - 2P_T - P_TP_G - P_TP_GP_B$. From this we deduce that the Gshare predictor will reduce the average misprediction penalty if $P_G > 2P_B \,/\, (1 + P_B)$. In other words, the Gshare predictor will be effective as long as it is significantly more accurate than the BTB.

BTB	Gshare	TAGE	Penalty
X	X	0	4
X	0	1	2
0	1	1	1
1	1	1	0

Table 3. Penalty with BTB, Gshare, and TAGE predictors

Microarchitectural Description

Pipeline

Speculative state in the pipeline is managed with the help of the PcUnit module. The PcUnit module holds the PC as well as the three boolean epoch counters. When the instruction flow is redirected, the relevant epoch counters in PcUnit as well as the PC for the instruction that is next to be fetched is updated. Ephemeral History Registers (EHRs) are used to eliminate rule conflicts and allow multiple redirections to occur within the same cycle. The EHRs are configured such that if multiple redirections occur within the same cycle, the redirection furthest down the pipeline takes precedence. The effects of the redirection are felt in the cycle after the redirection takes place. This is accomplished through the internal rule doRedirect which fires before all other rules in the same cycle. The interface for the PcUnit module is presented below.

```
interface PcUnit;
    method Word getPc();
    method Bool getEpochGshare();
    method Bool getEpochTAGE();
    method Bool getEpochEx();
    method Bool matchEpochGshare(Bool epGshare, Bool epTAGE, Bool epEx);
    method Bool matchEpochTAGE(Bool epTAGE, Bool epEx);
    method Bool matchEpochEx(Bool epEx);
    method Action updatePc(Word ppc);
    method Action redirGshare(Word rpc);
    method Action redirTAGE(Word rpc);
    method Action redirEx(Word rpc);
endinterface
```

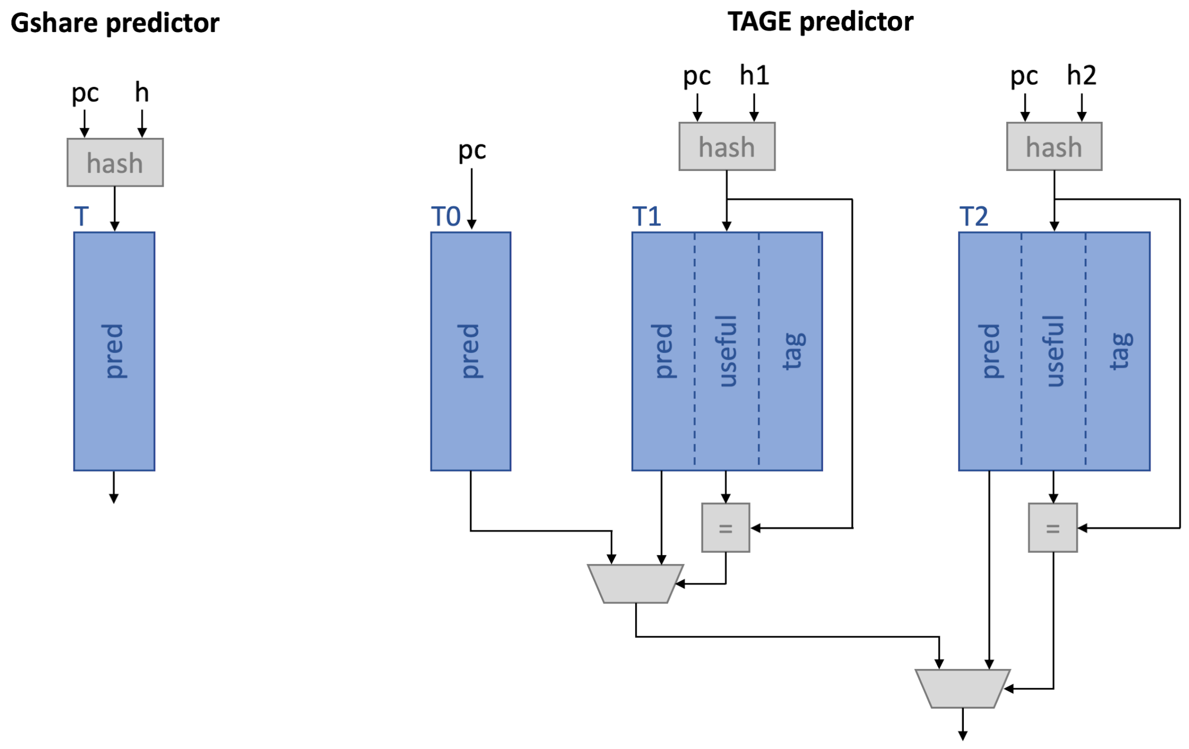


Figure 2. Gshare and TAGE branch predictors

Gshare predictor

The Gshare predictor uses an N-bit global history register to keep track of the direction of the last N branches. It maintains a branch history table (BHT) with 2^N 2-bit saturating counters. The global branch history is XORed with the PC to give the BHT index. In the execute stage, the counters are incremented if the branch was taken and decremented otherwise. An illustration of the Gshare predictor is shown in Figure 2 and its interface is presented below.

```
interface Gshare;
    method Bool getPred(Word pc);
    method Action train(Word pc, Bool taken);
endinterface
```

TAGE predictor

The TAGE predictor uses a series of M predictor tables, T_i with $0 \leq i < M$, each indexed with a different history length. This allows it to capture correlations from distant branches while allocating most of its memory to entries with short history length. Table T_0 is indexed with only the PC and provides the default predictions. The upper tables are partially tagged and make use of geometrically increasing history lengths. If a match is found in multiple tables, the table that uses the longest history length takes precedence. All of the upper tables have a 3-bit saturating prediction counter and a 2-bit saturating useful counter. If a branch is matched in multiple tables and the matching table with the longest history length T_i makes a correct prediction but the matching table with the next longest history length T_j does not, the useful counter in T_i is incremented. The useful counter serves as a hint for evictions when new entries are inserted into the table. For a full description of the update policy of the TAGE predictor, refer to the paper by A. Seznec [2]. An illustration of the TAGE predictor is shown in Figure 2 and its interface is presented below. The TAGE_trainData type holds information that is used by the update policy such as the index of the matching tables T_i and T_j.

```
interface TAGE;
    method Action putPc(Word pc, Bool epochTAGE, Bool epochEx);
    method ActionValue#(Tuple2#(Bool, TAGE_trainData)) getPred();
    method Action train(Word pc, Bool pred, Bool taken, TAGE_trainData trainData);
endinterface
```

Verification

Pipeline

We verify the correctness of the pipeline by running the processor on assembly and C benchmarks. The design passes all test cases in the benchmark suite, which gives us confidence that all wrong path instructions are being squashed and all right path instructions are being executed. We also inserted instrumentation printouts in the processor source code to record the state of the pipeline at every cycle. An example of such a trace dump is shown below. Note the redirection in cycle 5818. By inspecting these dumps we can verify certain aspects of the operation of the pipeline. For instance, that redirects take effect in the cycle after it is triggered, or that instructions marked as wrong path are appropriately discarded.

```
5816: [D2]   pc=0x00000480 valid=1 redir=0 ppcTAGE=0x00000484
5816: [EX]   pc=0x00000478 valid=1 iType=Alu mispred=0 nextPc=0x0000047c
5816: [D3]   pc=0x0000047c valid=1 hazard=1
5817: [D1]   pc=0x00000458 valid=1 iType=Alu redir=0 ppcGshare=0x0000045c
5817: [WB]   pc=0x00000478 iType=Alu
5817: [D3]   pc=0x0000047c valid=1 hazard=0
5818: [FE]   pc=0x00000460 epochGshare=1 epochTAGE=1 epochEx=0
5818: [D2]   pc=0x00000484 valid=1 redir=1 ppcTAGE=0x00000488
5818: [EX]   pc=0x0000047c valid=1 iType=Alu mispred=0 nextPc=0x00000480
5818: [D3]   pc=0x00000480 valid=1 hazard=1
> REDIRECT   redirPc=0x00000488 redirType=RedirTAGE epochGshare=1 epochTAGE=0 epochEx=0
5819: [D1]   pc=0x0000045c valid=0
5819: [D2]   pc=0x00000458 valid=0
5819: [WB]   pc=0x0000047c iType=Alu
5819: [D3]   pc=0x00000480 valid=1 hazard=0
5820: [FE]   pc=0x00000488 epochGshare=1 epochTAGE=0 epochEx=0
5820: [D1]   pc=0x00000460 valid=0
5820: [EX]   pc=0x00000480 valid=1 iType=Alu mispred=0 nextPc=0x00000484
5820: [D3]   pc=0x00000484 valid=1 hazard=1
5821: [FE]   pc=0x0000048c epochGshare=1 epochTAGE=0 epochEx=0
5821: [D1]   pc=0x00000488 valid=1 iType=Jr redir=0 ppcGshare=0x0000048c
5821: [WB]   pc=0x00000480 iType=Alu
5821: [D3]   pc=0x00000484 valid=1 hazard=0
5822: [FE]   pc=0x00000490 epochGshare=1 epochTAGE=0 epochEx=0
5822: [D1]   pc=0x0000048c valid=1 iType=Alu redir=0 ppcGshare=0x00000490
5822: [D2]   pc=0x00000488 valid=1 redir=0 ppcTAGE=0x0000048c
5822: [EX]   pc=0x00000484 valid=1 iType=Br mispred=1 nextPc=0x00000458
```

Predictor structures

To verify that our three predictors—BTB, Gshare, and TAGE—are functionally correct, we wrote software models that mirror the Bluespec implementations. The trace dumps from the benchmarks are fed into these models. Discrepancies between the trace dump and the model are flagged as errors. An example of the checker output with simulated errors is shown below.

```
Verifying test median
PASSED

Verifying test multiply
PASSED

Verifying test qsort
Error: tage prediction mismatch cycle=11503 pc=0x000005b8 rtl=1 model=0
Error: tage prediction mismatch cycle=11625 pc=0x000005b8 rtl=1 model=0
Error: tage prediction mismatch cycle=11676 pc=0x000005b8 rtl=1 model=0
FAILED
```

Implementation Evaluation

Utilization

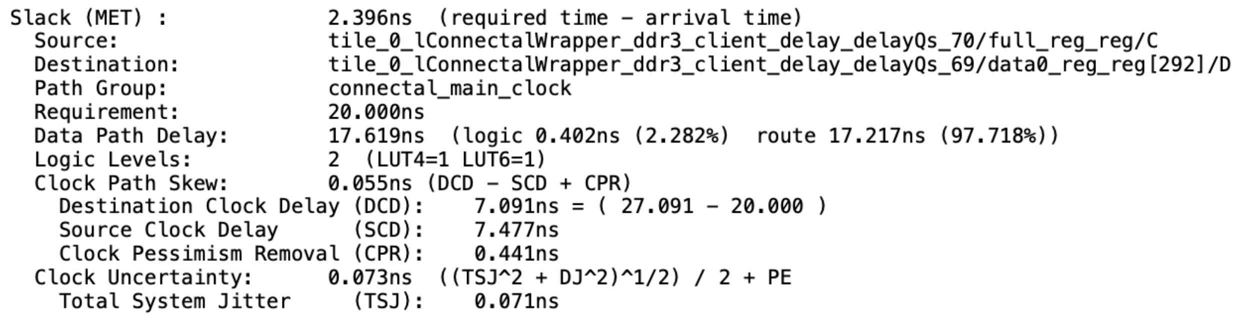
We compile our design for the Virtex-7 FPGA. The total LUT utilization of the design is 31.54%. However, most of this is consumed by the PCIe and DDR3 interface logic. The total LUT utilization of the processor is 7.33%, an 80% increase over the initial three stage bypass design. A breakdown of the processor resource utilization is given in Table 4.

Resource	**Utilization**
Total LUTs	22255 (7.33%)
Logic LUTs	22253 (7.33%)
LUTRAMs	2 (0.01%)
FFs	18546 (3.05%)
RAMB36	14 (1.36%)
RAMB18	6 (0.29%)

Table 4. Processor resource utilization

Timing

The timing report for the FPGA implementation is shown below. The critical path seems to be within the DDR3 interface logic. It only has a logical depth of 2, with 97.7% of the delay coming from the routing. The Yosys synthesis report indicates that the critical path within the processor is from the instruction cache to the PC redirection register.



```
Slack (MET) :             2.396ns  (required time - arrival time)
  Source:                 tile_0_lConnectalWrapper_ddr3_client_delay_delayQs_70/full_reg_reg/C
  Destination:            tile_0_lConnectalWrapper_ddr3_client_delay_delayQs_69/data0_reg_reg[292]/D
  Path Group:             connectal_main_clock
  Requirement:            20.000ns
  Data Path Delay:        17.619ns  (logic 0.402ns (2.282%)  route 17.217ns (97.718%))
  Logic Levels:           2  (LUT4=1 LUT6=1)
  Clock Path Skew:        0.055ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    7.091ns = ( 27.091 - 20.000 )
    Source Clock Delay      (SCD):    7.477ns
    Clock Pessimism Removal (CPR):    0.441ns
  Clock Uncertainty:      0.073ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
```

Performance

We evaluate the performance of our predictors on the big processor benchmarks. The benchmarks have been modified to repeat the computation such that they are each normalized to about 10,000,000 instructions. The performance benefits of the multi-tiered prediction structure is inconclusive. We expect that the BTB is least accurate at predicting branches, followed by the Gshare predictor and the TAGE predictor. However, that is not the observed behavior. There are benchmarks where each of the three predictors outperform the other two by a substantial margin. One plausible explanation for this behavior is that these branch predictors are tuned for realistic branch patterns, and the branch patterns in these benchmarks are not “realistic”. For instance, the qsort benchmark performs quicksort on a random array. The branches in this case would be entirely data dependent and almost completely random. Since there is no underlying correlation in the branch history, the Gshare and TAGE predictors both perform worse than the BTB.

The hit rate for the predictors across the benchmarks is presented in Table 4. This is evaluated for a 4-entry BTB, 256-entry Gshare predictor, and 5-component 64-entry TAGE predictor. The BTB makes predictions for all instructions, but we only quote the hit rate on branch instructions.

Benchmark	**BTB**	**Gshare**	**TAGE**
towers	214034 (66.9%)	292695 (91.4%)	284371 (88.8%)
median	2288566 (61.9%)	2835521 (76.6%)	2580350 (69.7%)
multiply	2004575 (60.0%)	2609500 (78.2%)	3004702 (90.0%)
qsort	1655807 (63.2%)	1586162 (60.6%)	1503270 (57.4%)
vvadd	1264422 (99.9%)	1264626 (99.9%)	1264664 (99.9%)

Table 5. Predictor hit rate across benchmarks

Design Exploration

We investigate the effect of table size on the performance of the predictors. Figure 3 shows the impact on the hit rate as we vary the side of the predictor tables from 4 to 256. While the general trend is that increasing the size of the table improves the hit rate, in many instances this is not true. This is probably because the contrived nature of the benchmarks do not reflect realistic processor behavior. For instance, the BTB saturates at 16 entries. Any further increase in table size does not affect the hit rate. The benchmarks that we have used only have about 16 distinct branches, much less than what a realistic program might have.

There is some anomalous behavior in the hit rates for the BTB and TAGE predictor. For the BTB, increasing the table size actually decreases the hit rate for the median and towers benchmarks. It seems that address aliasing actually improved the accuracy in this case. For the TAGE predictor, there is a dip in accuracy for the towers benchmark at $n = 32$ and the multiply benchmark at $n = 256$. This is probably due to the specific interaction of the loop size and the hash function that computes the index for the tables. Overall,

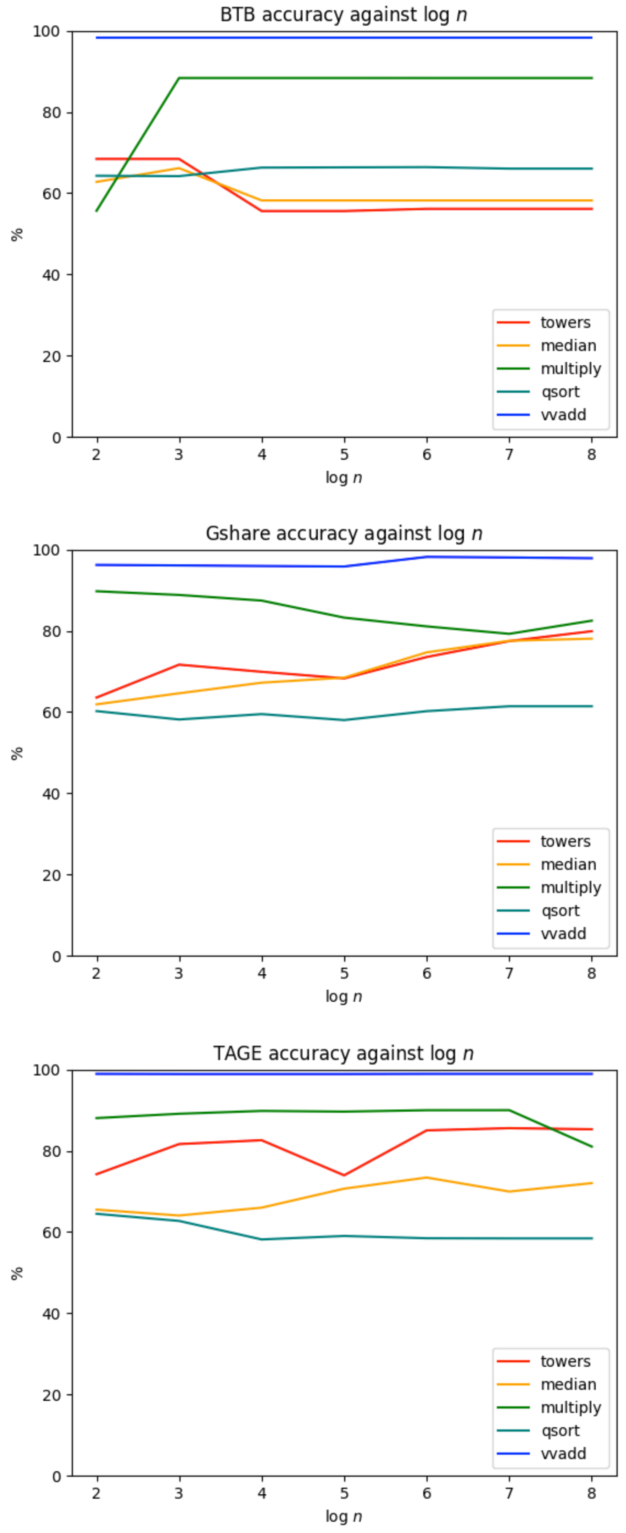


Figure 3. Predictor hit rate against size

it seems that benchmarks with more realistic program behavior are needed to meaningfully characterize the performance of these predictors.

Resources

The code for the project is hosted at https://github.mit.edu/frwang/6375-FP.

References

[1] McFarling S. (1993), *Western Research Laboratory Technical Note TN-36: Combining Branch Predictors*.

[2] Seznec A. and Michaud P. (2006), *A Case for Partially Tagged Geometric History Length Branch Prediction*, Journal of Instruction-level Parallelism 8.