Lab 3 Open Ended Section

Design Approach

First we looked at the different patterns that were found in the various benchmark tests. To be more specific we classified two different types of branching: local branching and global branching. Most of the time the benchmark tests had loops that had a predictable end. Other times, the code generally had paths that were commonly taken, like a series of if statements. We would base our predictor off of this observation.

The main approach we took was to separate the predictions into local and global. Then we would train a third predictor to decide which one to choose. For global inputs, we used the gshare algorithm and for local inputs, we used one developed by Yeh and Patt [1], also referenced in the spec. Essentially, local branching can be described by the recent branching history based on the branch address, and global branching can be described by the branch address and the current global branch history.

Implementation Challenges

- Deciding how to implement the different levels and deciding which algorithm to use for each case
- Deciding how many entries and how many bits we would use for the saturation counters
- Deciding what value to initialize the different counters to
- How to update the arbiter correctly when the global and local predictions differ
- Training the predictor to learn different global patterns, where we jump to different parts of the code

Overview of Algorithm

Variables needed:
Global Counter Table: 4096 entries of 2 bit saturating counters, indexed by global history of last 12 branches
Local History Table: 1024 entries of 10-bit branch patterns indexed by PC
Local Counter Table: 1024 entries of 3 bit counters
Arbiter: 4096 array of 2 bit counters, indexed by global history (12 bits), stores a 2 bit saturating counter

Helpers created:
local_prediction(**unsigned long long ip, unsigned long long hist, unsigned char *pred**)
global_prediction(**unsigned long long ip, unsigned long long hist, unsigned char *pred**)
arbiter_decide(**unsigned long long ip, unsigned long long hist, unsigned char *pred**)
local_update(**unsigned long long ip, unsigned long long hist, unsigned char taken**)
global_update(**unsigned long long ip, unsigned long long hist, unsigned char taken**)
arbiter_update(**unsigned long long ip, unsigned long long hist, unsigned char taken**)

Predict Algorithm: inputs(**unsigned long long ip, unsigned long long hist, unsigned char *pred**)
Local Prediction Algorithm:

- Get first 10 bits of ip and use that to index into the local history array to get local branch history
- Use local branch history to index the 3 bit counters, if < 4 then predict no, if >= 4 then yes

Local Prediction Code:

```
local_prediction(unsigned long long ip, unsigned long long hist, unsigned char *pred) {
  local_hist = LocalHistoryTable[ip[0:10]];
  local_counter = LocalCounterTable[local_hist];
  *pred = (local_counter >= 4) ? 1 : 0;
}
```

Global Prediction:

- Get first 12 bits of hist and use that to index into the global table.
- This value if < 2 predict no, if >= 2 predict yes

Global Prediction Code:

```
global_prediction(unsigned long long ip, unsigned long long hist, unsigned char *pred) {
  global_counter = GlobalCounterTable[hist[0:12]];
  *pred = (global_counter >= 2) ? 1 : 0;
}
```

Arbiter Decision:

- Index arbiter decision table using the 12 bits of global history, if < 2 then return no, if >= 2 return yes

Arbiter Prediction Code:

```
arbiter_decide(unsigned long long ip, unsigned long long hist, unsigned char *pred) {
 *pred = (Arbiter[hist[0:12]] >= 2) ? 1 : 0;
}
```

Update Algorithm: inputs(**unsigned long long ip, unsigned long long hist, unsigned char taken**)
Local Update Algorithm:

- Get first 10 bits of ip to index into the local history array
- Use that history to index into the counter table
- Add 1 to the counter if actual was yes, subtract 1 if no
- Then update the local history by shifting left and replacing least significant bit by taken

Local Update Code:

```
local_update(unsigned long long ip, unsigned long long hist, unsigned char taken) {
  local_hist = LocalHistoryTable[ip[0:10]];
  local_counter = LocalCounterTable[local_hist];
  if (taken) {
    LocalCounterTable[local_hist] = local_counter + 1;
  } else {
    LocalCounterTable[local_hist] = local_counter - 1;
  }
 LocalHistoryTable[ip[0:10]] = taken ? local_hist << 1 + 1 : local_hist << 1;
}
```

Global Update Algorithm:

- Get first 12 bits of global history, then use that to index into counter table
- Add 1 to counter if actual was yes, subtract 1 if no

Global Update Code:

```
global_update(unsigned long long ip, unsigned long long hist, unsigned char taken) {
  global_counter = GlobalCounterTable[hist[0:12]];
  int global_counter = GlobalCounterTable[global_hist];
  if (taken && global_counter < GLOBAL_MAX_COUNTER) {
            GlobalCounterTable[global_hist] = global_counter + 1;
  }
  if (!taken && global_counter > 0) {
            GlobalCounterTable[global_hist] = global_counter - 1;
  }
}
```

Arbiter Update Algorithm:

- Get first 12 bits of global history, then use that to index into arbiter table
- Determine the which type of prediction we chose, then update based on whether that prediction was correct or not
- Add 1 to counter if actual was yes, subtract 1 if no

Arbiter Update Code:

```
arbiter_update(unsigned long long ip, unsigned long long hist, unsigned char taken) {
  arbiter_counter = Arbiter[hist[0:12]];
  if (Arbiter[arbiter_hist] >= 2) {
          if (taken == pred && arbiter_counter < GLOBAL_MAX_COUNTER) {
          Arbiter[arbiter_hist] = arbiter_counter + 1;
          } else if (taken != pred && arbiter_counter > 0){
          Arbiter[arbiter_hist] = arbiter_counter - 1;
          }
  } else {
          if (taken != pred && arbiter_counter < GLOBAL_MAX_COUNTER) {
          Arbiter[arbiter_hist] = arbiter_counter + 1;
          } else if (taken == pred && arbiter_counter > 1) {
          Arbiter[arbiter_hist] = arbiter_counter - 1;
          }
  }
}
```

Branch Predictor Code:

```
#define INTERNAL_TABLE_SIZE 1024
#define EXTERNAL_TABLE_SIZE 4096
#define LOCAL_MAX_COUNTER 7
#define GLOBAL_MAX_COUNTER 3

#include <stdint.h>
#include <stdio.h>

int LocalHistoryTable[INTERNAL_TABLE_SIZE];
int LocalCounterTable[INTERNAL_TABLE_SIZE];
int GlobalCounterTable[EXTERNAL_TABLE_SIZE];
int Arbiter[EXTERNAL_TABLE_SIZE];

extern "C" void initialize_branch_predictor()
{
  for (int i = 0; i < INTERNAL_TABLE_SIZE; i++) {
          LocalHistoryTable[i] = 4;
          LocalCounterTable[i] = 4;
  }

  for (int i = 0; i < EXTERNAL_TABLE_SIZE; i++) {
          GlobalCounterTable[i] = 1;
          Arbiter[i] = 1;
  }
}

extern "C" void predict_branch(unsigned long long ip, unsigned long long hist, unsigned char
*pred)
{

  // Local Prediction
  unsigned long long ip_index = ((1 << 10) - 1) & ip; // Gets first 10 bits of ip
  unsigned long long local_hist = ((1 << 10) - 1) & LocalHistoryTable[ip_index]; // Gets first 10
bits of local history
  int local_counter = LocalCounterTable[local_hist];
  int local_pred = (local_counter >= 4) ? 1 : 0;

  // Global Prediction
  unsigned long long global_hist = ((1 << 12) - 1) & hist; // Gets the first 12 bits of hist
  int global_counter = GlobalCounterTable[global_hist];
  int global_pred = (global_counter >= 2) ? 1 : 0;
```

```
  //Arbiter Prediction
  unsigned long long  arbiter_hist = ((1 << 12) - 1) & hist; // Gets the first 12 bits of hist
  *pred = (Arbiter[arbiter_hist] >= 2) ? global_pred : local_pred;
}

extern "C" void update_branch(unsigned long long ip, unsigned long long hist, unsigned char
taken)
{

  unsigned char pred; // Stores the previous prediction
  predict_branch(ip, hist, &pred);

  // Local Update
  unsigned long long ip_index = ((1 << 10) - 1) & ip; // Gets first 10 bits of ip
  unsigned long long local_hist = ((1 << 10) - 1) & LocalHistoryTable[ip_index]; // Gets first 10
bits of local history
  unsigned long long local_counter = LocalCounterTable[local_hist];
  if (taken && local_counter < LOCAL_MAX_COUNTER) {
        LocalCounterTable[local_hist] = local_counter + 1;
  }
  if (!taken && local_counter > 0) {
        LocalCounterTable[local_hist] = local_counter - 1;
  }
  LocalHistoryTable[ip_index] = taken ? (local_hist << (1 + 1)) : (local_hist << 1);

  // Global Update
  unsigned long long global_hist = ((1 << 12) - 1) & hist; // Gets the first 12 bits of hist
  int global_counter = GlobalCounterTable[global_hist];
  if (taken && global_counter < GLOBAL_MAX_COUNTER) {
        GlobalCounterTable[global_hist] = global_counter + 1;
  }
  if (!taken && global_counter > 0) {
        GlobalCounterTable[global_hist] = global_counter - 1;
  }

  // Arbiter Update
  unsigned long long arbiter_hist = ((1 << 12) - 1) & hist; // Gets the first 12 bits of hist
  int arbiter_counter = Arbiter[arbiter_hist];
  if (Arbiter[arbiter_hist] >= 2) {
        if (taken == pred && arbiter_counter < GLOBAL_MAX_COUNTER) {
        Arbiter[arbiter_hist] = arbiter_counter + 1;
        } else if (taken != pred && arbiter_counter > 0){
```

```
        Arbiter[arbiter_hist] = arbiter_counter - 1;
        }
  } else {
        if (taken != pred && arbiter_counter < GLOBAL_MAX_COUNTER) {
        Arbiter[arbiter_hist] = arbiter_counter + 1;
        } else if (taken == pred && arbiter_counter > 1) {
        Arbiter[arbiter_hist] = arbiter_counter - 1;
        }
  }
}
}
```

Benchmark Outputs:

CPI Table:

CPI = mcycle/minstret	dhrystone	median	multiply	qsort	spmv	towers	vvadd
Rocket	1.200	1.379	1.136	1.427	1.690	1.029	1.024
BOOM (original branch predictor)	187545/196029 = 0.957	9719/4156 = 2.339	27574/24101 = 1.144	234517/127368 = 1.841	36522/34851 = 1.048	5103/6170 = 0.827	1638/2417 = 0.678
BOOM (modified branch predictor)	174360/196029 = 0.889	9237/4156 = 2.223	26602/24101 = 1.104	262769/127368 = 2.063	35645/34851 = 1.023	3898/6170 = 0.632	1641/2417 = 0.679

Branch Prediction Accuracies:

	dhrystone	median	multiply	qsort	spmv	towers	vvadd
BOOM (original branch predictor)	143/7340.0 jalrs, 0.019	34/77.0 jalrs, 0.442	37/279.0 jalrs, 0.133	38/86.0 jalrs, 0.442	36/81.0 jalrs, 0.444	35/331.0 jalrs, 0.106	35/75.0 jalrs, 0.467
[Mispredicts by type]	1088/32720 brs, 0.033	1042/4753 brs, 0.219	1509/14325 brs, 0.105	11050/61875 brs, 0.179	1621/7281 brs, 0.223	239/1635 brs, 0.146	67/1976 brs, 0.034
	1231/40060.0 all, 0.031	1076/4830.0 all, 0.223	1546/14604.0 all, 0.106	11088/61961.0 all, 0.179	1657/7362.0 all, 0.225	274/1966.0 all, 0.139	102/2051.0 all, 0.050
BOOM (modified branch predictor)	145/7340.0 jalrs, 0.020	35/77.0 jalrs, 0.455	37/279.0 jalrs, 0.133	38/86.0 jalrs, 0.442	37/81.0 jalrs, 0.457	36/331.0 jalrs, 0.109	34/75.0 jalrs, 0.453
[Mispredicts by type]	637/32646 brs, 0.019	1059/4749 brs, 0.223	1358/14333 brs, 0.095	13750/61893 brs, 0.222	1619/7299 brs, 0.222	131/1653 brs, 0.079	83/1982 brs, 0.042
	782/39986.0 all, 0.020	1094/4826.0 all, 0.227	1395/14612.0 all, 0.095	13788/61979.0 all, 0.222	1656/7380.0 all, 0.224	167/1984.0 all, 0.084	117/2057.0 all, 0.057

Performance Summary

- The CPI improved in most cases, with a few exceptions of functions that were dependent on global branching. For instance, Qsort's paths are sometimes unpredictable without any compiler or software hints, because the branches depend on the data, which we don't have access to.
- The Branch Prediction miss rate varies, in some cases beating the bimodal system by quite a bit, while in others falling behind a little. The predictor is very good at predicting local branching, for example when a branching pattern repeats. This is why for loop heavy tests like dhrystone and multiply beat the original branch predictor. We do worse when trying to predict the global path, like with Qsort where the path will most likely be different ever run.

Questions

- How did you calculate the amount of state your branch predictor has?

  We take our counter bits and multiply by the number of entries in each array. We have (1024 * 3 * 2 + 4096 * 2 * 2) bits = 22528 bits = 2.8 Kb of state.

- How do certain parameters (e.g., number of entries) impact accuracy?

  - Number of entries: The more entries we have, the more sparse our array would be, however it would reduce aliasing meaning we would see an increase in accuracy
  - Number of state bits for saturation counter: If the saturation counter has a low number of bits, the predictor would change predictions more often. This increases accuracy when the branch pattern switches often.

- Which branches or patterns were easier or harder to predict?

  Repetitive for loops were easier to predict while code with more conditional statements was harder to predict.

- What kind of application code do you expect your predictor to perform better or worse on?

  For loops and while loops is where we expect it to perform better, while it will perform worse on random jumps and chains of conditional statements.

- Do you foresee any challenges with the implementation of your predictor algorithm as a hardware block within a superscalar, out-of-order core?

  The biggest challenge we faced was how to know if a branch was uncommitted or not. Predict() is always run even if the branch can be kicked out of the pipeline, so we decided that it was best to recalculate the prediction again, then update the table.

- What changes or alternative approaches would you pursue as future work if more time were available?

  We would have created a perceptron or neural net that can learn from the patterns and learn the model and apply the best prediction each time.

References

[1] S. McFarling, Combining Branch Predictors, Tech. Note TN-36, Compaq Computer Corp. Western Research Laboratory, Palo Alto, Calif., June 1993