ECE 552 Lab 2 Report
Rudy Jin & Yuhe Chen
– Oct 20, 2023

Microbenchmark

- Loops and function calls are added to intentionally create PC address jumps.
- Complexity in loop if & while loop to trick two-level predictor; fixed iteration loop as well. This is to create various possible conditions for branching.
- Assembly code for main function and part of modify_values are provided to demonstrate the PC address change. (rand() is too long to describe in assembly)
- I used -01 -S flag.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int modify_value(int a) {       // I created this function to create a return instruction for testing. Also it uses rand() inside which adds certain complexity.
    return a + (rand() % 5);    // A lot of lines for assmebly code.
}                               // Ended with lw $31,20($sp); lw $16,16($sp); addu $sp,$sp,24; j $31; .end modify_value


int main() {
    // srand(time(NULL));          // Add certain degree of randomness if needed.
    int a = 10000;
    while (a > 5) {             // A loop to add conditional branches.
        a = a / 3;
        a = modify_value(a);    // Call function for PC to jump to address.
    }
    int b = 2;
    for (int i = 0 ; i < 100; i++){   // Fixed Upperbound Conditional branches
        if (b * a > 32) {             // Add complexity to two-level predictor state
            break;
        }
        a = a - b;
        b = b + 1;
        a = (a > 0) ? a : a * (-1); // Software aspect not to let a < 0 to satisfy if condition
    }

    return 0;

}
/* Put it side by side so the screenshot would not take too much space in report.
main:                                   modify_value:
        .frame $sp,24,$31                       Presented above next to C code for better understanding.
        .mask  0x80000000, -8
        .fmask 0x00000000,0
        subu   $sp, $sp, 24
        jal    __main
        move   $4, $0
        jal    time
        move   $4, $2
        jal    srand
        lw     $31, 16($sp)
        addu   $sp, $sp, 24
        j      $31
        .end   main
*/
```

Figure 1. Microbenchmark Code Review

Open-ended Branch Prediction

Our solution reaches 7.29% misprediction in average. However, in the last minute, I found out that I used double as storage, even if it is unnecessary… But that exceeds to limit of 128K bytes.

We created a perceptron predictor of size 2048*50 with a global history table of 50 bits and a 2048 * 2bit_saturation predictor with a 1024*11 table. The GHR records all branching Taken/Not Taken behaviors, which is used to (1) last 11 bits indexed into perceptron predictor entry (2) all 50 bits used to dot product with weights; the table for two- bit saturation predictor is indexed by 11 bits from one row of the table, which is selected by PC bits. Another integer competence record which predictor is better. When larger than 0, perceptron, vice versa. If 0, highest correction rate predictor. So the total is 2048*50 + 50 + 2048*2 + 1024*11 = 118K, :) However, I used double for weight perceptron model, so first term is replaced by 2048 * 50 * 64

	Two Bit Sat	Two Level	Perceptron & Two Bit
Astar	3695923 24.639%	1785464 11.903%	654452 4.363%
Bwaves	1181950 7.880%	1071909 7.146%	732787 4.885%
Bzip2	1224989 8.167%	1297677 8.651%	1261320 8.409%
GCC	3161205 21.075%	2223671 14.824%	530953 3.540%
Gromacs	1361054 9.074%	1122586 7.484%	1013699 6.758%
Hmmer	2035059 13.567%	2230774 14.872%	2136392 14.243%
MCF	3657995 24.387%	2024172 13.494%	1640933 10.940%
Soplex	1066132 7.108%	1022869 6.819%	778926 5.193%
Microbenchmark	1908 16.223%	1812 15.407%	1840 15.645%

Table 1. Result of Benchmark Performance under Various Predictors

Hardware Performance

Two Level - PureRAM: Each row demonstrates one table (512 * 1 + 64 * 8 * 1) Bytes
int and string are only for computation convenience; it use 1-bit representation in hardware.

Table Size	Access Time (ns)	Area (mm^2)	Leakage Power (mW)	Parameter Modified
64 * 8 * 1	0.164	1.05×10^{-3}	0.195	Size = 512
512 * 1	0.164	1.05×10^{-3}	0.195	Size = 512

Opened - PureRAM: (50 * 1 + 50 * 2048 * 32 + 1024 * 11 + 2048 * 2) = 3.29 bits (in bits because table below requires rounding to simulate).

Table Size	Access Time (ns)	Area (mm^2)	Leakage Power (mW)	Parame Modified
50 * 1	0.118	1.91×10^{-4}	0.030	Size = 64 (Has to be >= 64 to simulate)
2048*200	0.206	3.605×10^{-3}	0.834	Size = 2048; block = 200
1024 * 2	0.201	3.50×10^{-3}	0.834	Size = 2048; block = 2
2048 * 1	0.206	3.50×10^{-3}	0.834	Size = 2048; block = 1

Contribution:

We did the assignment (coding, debugging) together, evenly distributed.