HW2 REPORT

謝翔丞, 109550025

***Abstract*—本篇主要以兩個部分實行，首先是針對 branch history table 進行分析，嘗試不同 entry size number 以及 branch predict unit 的有無對於程式執行之影響，同時針對不同 branch 狀況也進行分析統計，發現一旦 table size 大於單次 iteration 中 branch 指令出現次數，即會達到趨近表現優化的極限; 第二部分是自行實作一個 two-level predictor ，並嘗試利用此 predictor 來優化整體的 performance。結果呈現 two level predictor 並非所有情況都能造成表現上升，反而可能因為考慮冗贅資訊導致 index 配對率大幅下降。**

Keywords—branch predictor , BHT, branch history table , Two level predictor (keywords)

I. INTRODUCTION

本次作業可拆分為兩階段，第一階段先針對現有 branch predict unit 進行嘗試開啟/關閉以及調整不同 entry size number (也就是 branch predict table 大小) 並觀察、分析現象以及不同 type 的 branch 之 hit / miss rate; 第二階段則是自行實作出一個 two-level 的 branch predictor ，嘗試以此實現整體效能的優化.

II. BRANCH ANALYSIS 前置作業與構思

A. 初步想法

觀察不同 entry size 導致的 performance 變化，我打算利用 coremark 執行後計算出的 Iteration / sec 來做比較，照我初步的推論，執行 performance 應會隨著 entry size 下降而隨之下降，因為這表示 table 能儲存的組數越少，也就代表預測上更容易遇到 table 上不存在的案例，造成需要不斷置換 table content、耗費操作及時間;反之也會隨著 entry size 上升而促使 performance 優化，但是達到某個上限之後表現會趨緩甚至可能持平，而這個上限就是每次 Iteration 內會遇到的 branch 指令次數，一旦全部會遇到的案例都能夠被放入 table 內部，那麼就不會有需要置換或是查找不到預測值的問題，也就表示假如 (branch instruction / iteration) = x 而 entry size = y 且 x < y 的話，對於任何 entry size z > y，其表現都與 entry size = y 相差無幾。以上為我個人初期的推論，詳細結果與驗證會在後續提及與說明。

B. 電路實作

要調整 entry size 有幾個地方要修改，除了在 bpu.v 內將以 ENTRY__NUM 設置的 addr_hit_CPU case 選項都更改成希望的 entry size 例如 32 等等之外，還要將 auila_top 中呼叫 core_top module 以及 core_top 呼叫 bpu module 時的參數也一併更改 .

```
// In auila_top.v
core_top#( .HART_ID( HART_ID ),  .XLEN( XLEN ),  .BPU_ENTRY_NUM( 32 ))
// In core_top.v
module core_top #(
    parameter HART_ID             = 0,
    parameter XLEN                = 32,
    parameter BPU_ENTRY_NUM  = 32  // <= here
// In bpu.v
case (addr_hit_PCU)
    32'h0000_0002: read_addr <= 1 ;
```

另外一部分是欲關閉 BPU 有幾個值要設定，分別是先在 core_top.v 設定代表 disable 的參數為 0，並在 program_counter.v 與 pipeline_control.v 將原先的 `ifdef ENABLE_BRANCH_PREDICTION 修改為該參數，以我為例便是更改如下圖:

取代 `ifdef ENABLE_BRANCH_PREDICTION

```
// In core_top.v
`define disable_bpu 0;
// In program_counter.v
`ifndef disable_bpu
// In pipeline_control.v
`ifndef disable_bpu
```

Analysis of Different Entry Size

以下是我針對不同 BPU 情形進行的分析結果，可以分為幾點:

C. *BPU 的使用與否對 Performance 之影響*

由 Figure 1 可見，.一旦關閉 BPU 功能，每單位時間內能完成的 iteration 次數會大幅下降，也就表示無法藉由預測下步執行位置來達到加速的效果; 相較之下，一旦使用 BPU 可以有效地提升單位時間內的 Iteration 次數，這是因為藉由 BHT 的紀錄我們可以先行猜測該指令要跳到何處，即使猜錯目的地，頂多也是變成和沒使用 BPU 一樣的執行過程，因此並不會造成增加 cycle 的反效果。

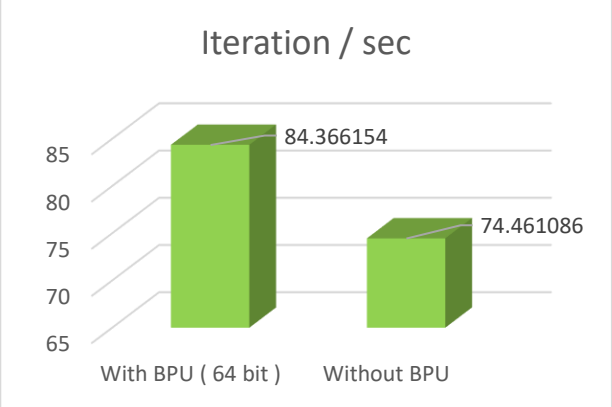

Figure 1.Iteration / sec Performance Comparison of With and Without BPU

D. *不同 Entry Size 之 BPU 對 Performance 之影響*

參照 Figure 2，可以發現單位時間內能完成的 Iteration 數量自從 Entry Size 超過 32 bit 之後就有逐漸緩和的趨勢，這符合在 II.A 初步想法 提及的猜測，一旦 Entry Size 增加到某個臨界值，整體表現就會因為 table 已能容納所有案例而逐漸趨緩; 反觀在 16 bit – 32 bit 這個 range 內有較大的落差，因此我猜測 Coremark 在每次 iteration 內所使用到 branch instruction 的次數可能在 16-32 之間。

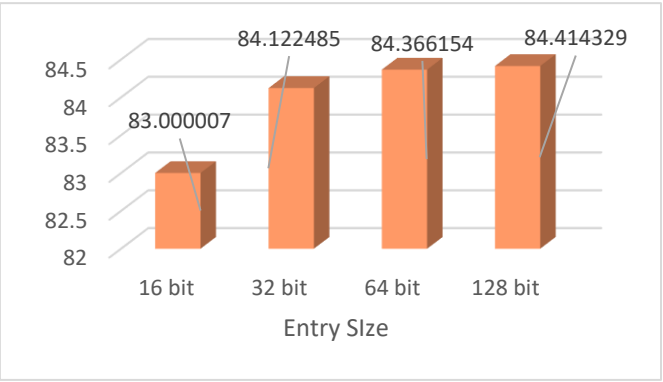

Figure 2.Iteration / sec Performance Comparison of Different Entry Size

E. *不同 Entry Size 之 hit / miss rate 比較*

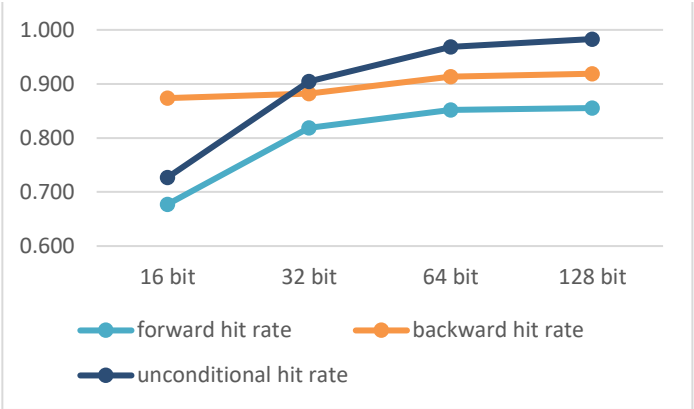

Figure 3.Hit / Miss Rate of Different Entry Size

F. *不同 Branch Type 之 hit / miss rate 比較*

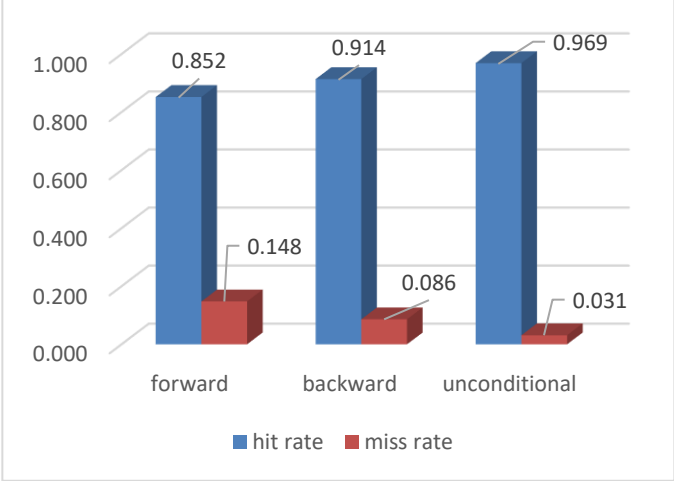

Figure 4.Hit / Miss Rate of Different Branch Type

III. TWO-LEVEL BRANCH PREDICTOR

A. *初步概念.*

原始 2 bit predictor 有它的限制在於它只 local branch prediction 導致疏漏 global 的資訊，我們可以嘗試實作 2 level branch predictor，差別在於並不是直接拿 address 當作 input，而是利用 Global Branch History 與 address 做 XOR 來作為 predictor 的 input 。我預計此次作業將使用講義第二種 predictor 的方式，將 dec_pc_i 的其中 6 bit 與 Global Branch History 做 XOR，並且這個 Global Branch History 值是由每次的 taken (1 bit)進行 concatenate 並取末 6 bit 的組合。上述概念化簡為圖可參考 Figure 5

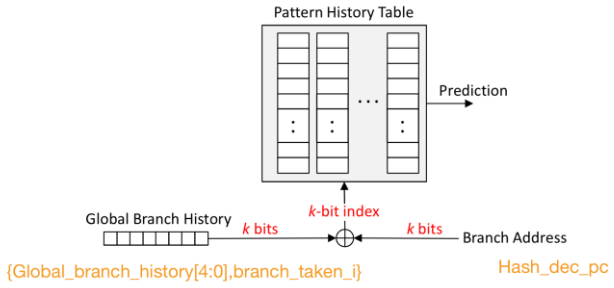

Figure 5.Two level predictor (2nd)

B. 電路實作

依照 Figure 5.Two level predictor (2nd)，我另外設置 6 bit 的 global_branch_history 以及 hash_pc、hash_dec_pc。

Hash 的方式是由如下:

```
hash_dec_pc <= (global_branch_history^dec_pc_i[8:3]);
```

(對於要取哪 6 bit 後續有詳細實驗)

而原先 branch_likelihood 要放入的 update_addr，改成上述 XOR 過後的值:

```
// branch_likelihood[update_addr] <= 2'b01;
branch_likelihood[hash_dec_pc] <= 2'b01;
```

C. 實驗結果

我首先嘗試使用 dec_pc_i[5:0] 作為 XOR 的對象。在我原先的猜測，two level predictor 應該要比原始的 saturating predictor 擁有更好的 performance 憶及更高的 iteration/sec ; 但結果 Figure 6 出乎意料的表現下降不少，因此我又嘗試各種不同 dec_pc bit 的組合，而數據**錯誤! 找不到參照來源。**顯示即使這些組合裡最好的表現(選用 dec_pc_i [6:1]作為 XOR operand)依舊遠遠不及原始 predictor 帶來的成果，因此我對這樣的數據進行一些分析與推論。

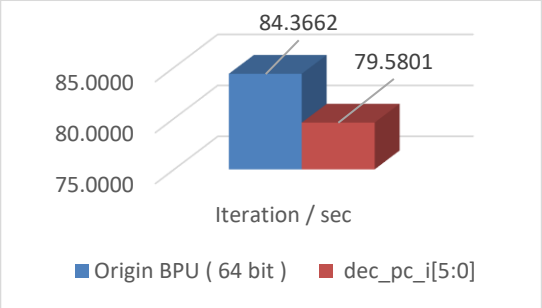

Figure 6.Performance Between Origin Predictor and Two Level Predictor

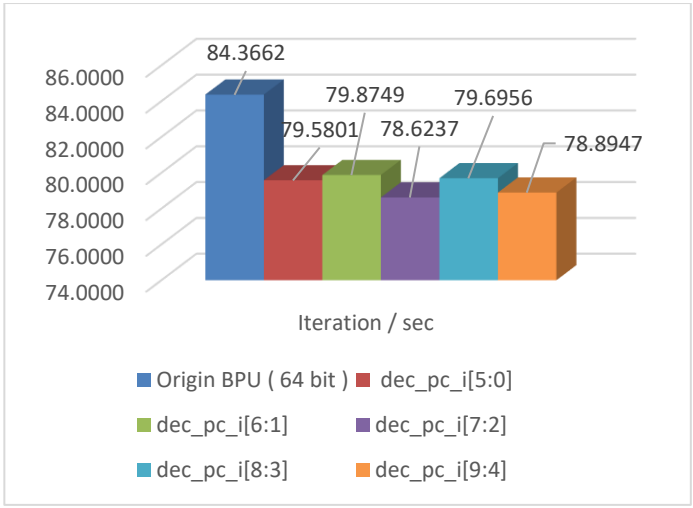

Figure 7.Performance Between Different Bit Combination

D. 分析與討論

針對 III.C 現象的成因，我有以下推論:

Coremark 執行過程較少或是不需要考慮 Global Data。若是 Coremark 其實只需將 local data 納入考量，那麼我實作的 Two Level Predictor 反而是將不必要的資訊考慮進去，複雜化的 index 導致 branch table 的 match 率大幅下降 。

當然，與 Figure 1 呈現 without BPU 的數據相比，該 Two Level Predictor 仍然有表現上的提升，代表此預測器仍有一定效力，只是相對單純使用 local branch history 的預測器反而考慮冗贅資訊 index 的 hash 致使配對率下降。

E. 結論

在某些情況下，Two Level Predictor 因為多考慮到 Global Branch History，使 table 能將更多資訊納入範圍，同時避免 likelihood 輕易地被改變 ; 但在面對只須考慮 local data 的狀況下，這個做法反而造成累贅，導致 index 更難被配對 table 必須不斷更新，重複移除與塞入不同值作為 index 致使 cycle 數量往沒 predict 的表現靠攏。