HW#2 Report
Return Address Predictor Design

邱振源, 111550100

***Abstract*—這次的做作業主要是針對 Aquila 的 BPU 做進行分析，找出關閉 BPU 或改變 BHT size 對不同 branch type 的 hit/miss rate 造成的影響。第二部分將實作 return address prediction(RAP)，並探討這樣的設計對 Aquila performance 的影響，同時探討改進 RAP 的方式。**

Keywords—RISC-V; Aquila; CoreMark; branch predict unit; profile circuit; BHT; return address prediction(RAP); TOS

I. INTRODUCTION

這次的作業主要是透過與 Lab1 類似的方法分析 RISC-V 處理器 Aquila 的電路狀況，藉由關閉 BPU 或是改變 BHT 的大小來觀察不同 branch type 的 hit/miss rate 的變化。而在第二部分則要設計並實作一個 return address predictor。

II. BRANCH PREDICTION ANALYSIS

A. *Profile Concept*

在觀察完 instruction 在 Aquila 的行為之後，我發現 Aquila 會藉由比對 BPU 的預測結果在 execution stage 進行 branch miss 的判斷，所以我將 profile 電路加在 execution.v 裡面。透過 is_branch_i 與 pc_i 和 branch_target_addr_o 的大小可以判斷是否為 conditional forward/backward jumps，再透過 is_jal_i 判斷目前是否是 unconditional jumps。

在判斷 miss predictions 的部分，branch_misprediction_o 是在 execution stage 藉由 branch_taken_o 與 decision 的比較再和 branch_hit_i 進行 and operation 而得，透過比較預測結果，或是 pc 有沒有 hit 到 BHT，求得不同 branch type 的 hit/miss rate。

B. *Disable BPU And Change BHT Size*

觀察整個 Aquila 之後，我發現只要修改 aquila_config.vh 中的`define ENABLE_BRANCH_PREDICTION，將其註解，就可以關閉 Aquila 對 BPU 的使用，從而使 Aquila stall 兩個 cycle，等待 execution stage 算出結果之後再繼續後面的 instruction。

而在改變 BHT size 的部分，可以透過修改 bpu.v 中的參數 parameter ENTRY_NUM 的數值，便可以改變整個 BHT 的 entry size，其他 BHT size 相關的參數也會因為改變 ENTRY_NUM 而跟著修正。

III. RESULT AND ANALYSIS

A. *Effect Of BHT*

在探討整個 BPU 對於處理器運作的影響，我們可以先觀察 BPU on/off 對運作 cycle 數的差異，因為在 disable BPU 之後，我發現 CoreMark 的 iteration 次數只能達到 1100 次，因此同樣比較相同迭代次數結果如下表(TABLE I)。

TABLE I. BPU ON V.S. BPU OFF

performance	BPU Off	BPU On
Iteration/Sec	89.021763	98.693354
Total Cycle	683528951	616926332
Branch Cycle	66271558	66490298
Stall Cycle	4782793	4782793

TABLE I 中 BPU On 的數據是將 ENTRY_NUM 設為 16 的結果，根據 TABLE I 可以發現，在開啟 BPU 之後，總 cycle 數下降，iteration/sec 上升，意味著 performance 在開啟 BPU 之後有顯著的提升，因為倘若關閉 BPU，便無法藉由預測 instruction 來減少 stall cycle 的產生，而使的總 cycle 數上升。

B. *Branch Type and Hit Rate*

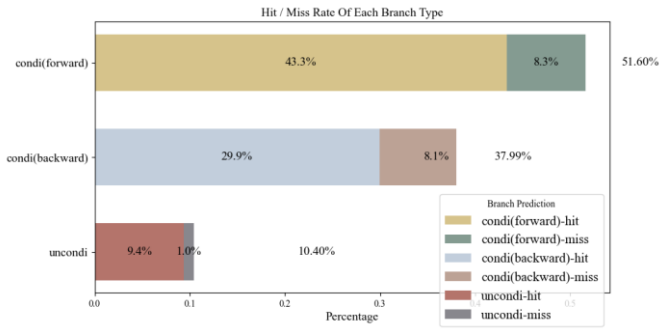

Fig. 1. Hit/Miss Rate of Each Branch Type

Fig. 1 是在 ENTRY_NUM 設定成 64 的時候，各種 branch type 所占有的比例與其 hit rate 與 miss rate 的長條圖，可以發現在 CoreMark 中 conditional forward jump 比例較高，而 unconditional jump 的 miss rate 較低，推測其在 BHT 記錄到的狀況可能較具規律性，而有較低的 miss rate。

C. *Analysis The Hit Rate of Each Entry Size*

TABLE II. THE HIT RATE OF EACH TYPE BY DIFFERENT ENTRY NUM

ENTRY	forward jump	backward jump	uncond jump	total
16	81.34%	60.96%	47.43%	70.13%
32	88.56%	72.54%	76.69%	81.24%
64	89.88%	78.79%	90.13%	85.69%
128	91.16%	84.82%	95.12%	89.16%
256	92.68%	87.27%	98.18%	91.20%
512	92.76%	87.40%	99.79%	91.45%

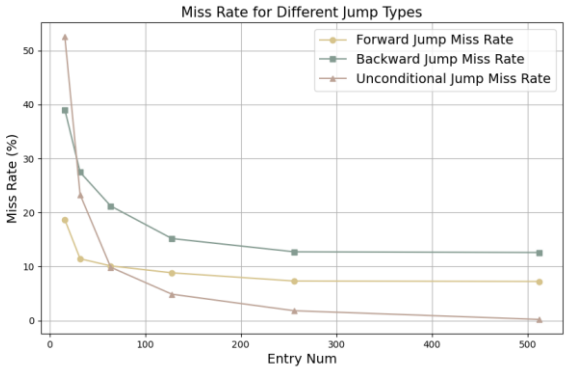

Fig. 2. Miss Rate for Different Jump Type

TABLE II 紀錄了改變 ENTRY_NUM 之後對三種不同 branch type 的 hit rate 影響，可以發現隨著 ENTRY_NUM 增大，hit rate 也有所提升，然而成長幅度逐間趨緩。同時，比較 Fig. 2.將不同 branch type 的 miss rate 化成折線圖的結果，同樣可以發現在一開始，miss rate 下降的幅度較大，而之後就逐漸趨緩。

這樣的現象可以透過 Amdahl's Law 來解釋，同時也可以推測在 CoreMark，用 256 地 BHT size 已經可以記錄到大部分 branch instruction 的 address，所以在節省空間的前提下 ENTRY_NUM 設為 256 對於 CoreMark 來說可能已經夠用了。

分析 TABLE II 中三種不同 branch type 的 hit rate 漸進極值可以發現 unconditional jump 的極值可以快趨近 100%，而 conditional forward jump 與 conditional backward jump 則分別是 92%與 87%，這樣的原因可能是因為 unconditional jump 對於 branch predict 的預測較為單純，只需要考慮 taken 就可以了，而 conditional jump 就需要預測在不同情況下是否要進行 taken，預測也就變得較為困難，hit rate 的極值自然較低。

IV. Architecture For Return Address Prediction

這次作業的第二部份是要實作 return address prediction，因為目前 Aquila 並沒有對針對這樣的 branch 進行 predict，而透過設計一個 return address stack (RAS)便可以讓我們再遇到 ret instruction 的時候，不必等到 execution stage 才算出要 return 回去的地方，這樣的方式可以在遇到大量的 function call 的時候有效的減少 stall cycle。

A. *Architecture of RAP*

一開始我參考老師在簡報上的 minimal implementation of RAP (Fig. 3.)，建立一個 return address predictor (RAP)，並將 is_ret 和 is_jal 和 current pc 傳入 RAP。

而在 RAP 內部，主要可以分成兩部分，首先是 ret HT，這個 history table 主要紀錄是 ret instruction 的 pc address，而我也仿照 BPU 用 direct map 的方式建立 ret HT，用部分的 pc address 作為 index，而該 pc address 作為 tag。第二部份社 RAS，我用一個 circular stack 來完成 RAS 的設計，因為 circular stack 會優先取代 stack 最底部的位置，也可能時最晚會被 pop 出來的，這樣就可以解決 overflow 的問題，而 RAS 的值是前一個 jal instruction 的 pc+4，這樣就可以直接回到 function call 完的地方接著執行。

而每次要做的事情就是在遇到 jal instruction 的時候將 pc+4 push 到 RAS，而在遇到 ret instruction 的時候，如果 ret HT miss 就將當前的 pc 加入 ret HT，如果 ret HT hit 就將 RAS 的 top 的值傳給 program counter，便可以完成簡單的 RAP。

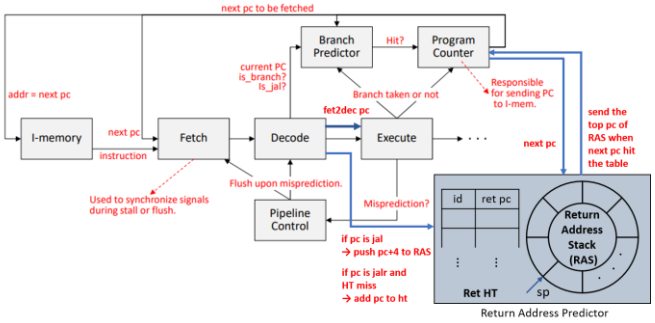

Fig. 3. Minimal Implementation of RAP

B. *Detail of Implementation*

首先是處理 misprediction 的部分，因為透過 RAP 預測的 target address 會在下一個 cycle 透過 program counter 進入 fetch 的階段，所以當我們要在 execution stage 處理 RAP misprediction 的問題的時候，target address 會在 decode stage，所以，我將 decode stage 的 pc 傳入 execution stage，並判斷 RAP 的 target address 是否跟在 execution stage 算出來的結果一致，若兩者不同，則透過 pipeline control 進行 flush。

而在處理 flush 的部分，因為原本的 branch_flush 沒有考慮到 RAP hit 的情況，所以我在原本的布林運算式裡面加了!rap_hit，來避免 Aquila 在 RAP 預測到的情況下仍然進行 instruction flush。而在處理 RAP 預測錯誤的 flush，則是在傳去 fetch 和 decode stage 的 flush 訊號加上 rap_mispredict 的判斷。

另外再處理 ret 訊號的不分，我原本是用 jalr 來直接判定成 ret 的訊號，然而，在後來看了.objdump 檔後發現，jalr 不只用在 function return 的地方，function pointer 呼叫的時候同時也會使用 jalr，如果將這類指令也判定成需要 pop RAS 的話可能會使的整個 RAP 因為 pop 失誤而讓整個 RAS 大亂，因此，我將 is_ret 的訊號直接指定為 ret instruction (0x0008067)。

C. *Decide The Ret HT Size*

TABLE III. The Performance of Different Ret HT Size

ENTRY	Iteraion/Sec	Total Cycle	Mispredict	HT Miss
RAP Off	101.198059	1540516618	X	X
32	101.724355	1532565211	5057479	118293
64	101.726011	1532540332	5073040	87184
128	101.728494	1532503020	5122803	12546
256	101.730150	1532478146	5122806	105
512	101.730150	1532478122	5122806	87

TABLE III 紀錄了在不同 ret HT size 下的 performance 情形(BHT 的 entry size=256)，在加入 RAP 之後 iteration/sec 約增加 0.5，而同時可以發現 performance 隨著 HT size 大而逐漸上升。然而，在 ret HT 的 entry size 為 256 和 512 的時候，iteration/sec 已經看不出差別，ret HT 的 miss 狀況也從數萬跌到 105，因此可以判斷，ret HT 的 entry size 在執行 CoreMark 的時候，設為 256 就已經很夠用了。

D. Decide The RAS Size

TABLE IV. THE PERFORMANCE OF DIFFERENT RAS SIZE

ENTRY	Iteraion/Sec	Total Cycle	Mispredict	HT Miss
2	101.730150	1532478140	5122803	105
4	101.730150	1532478146	5122806	105
8	101.730150	1532478146	8122806	105
16	101.730150	1532478149	5122806	105

接著，我想決定 RAS 的大小，可以看到，在 ret HT 射程 256 的狀況下，我試著將 RAS 的 entry num 設成 2, 4, 6, 8，然而，這些結果大致上看不出來太大的不同，可以推測 CoreMark 的 function call 的深度可能為 2~4 層。不過，由於下一階段實作部分的考量，我最終還是將 RAS 的 entry size 設為 4。

V. TOP-OF-STACK POINTER REPAIRING

在 improve RAP minimal implementation 的部分，我參考了一篇 Princeton University 的 paper，探討 Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms [1]，題中提到在執行 while 之類的程式碼的時候，可能會因為 branch take/flush 的問題，導致 RAP 預測錯誤。針對這個問題，paper 提出可以透過儲存 top-of-stack (TOS) pointer 來解決，只要先儲存住當前被 pop 之前的 RAS top pointer，便可以在發現 RAS 被錯誤 pop 的時候復原 stack top 的資訊，而讓因為 branch taken/flush 造成問題的 RAP mispredict 問題得到一個解決的方法。

A. Architecture of RAP with TOS

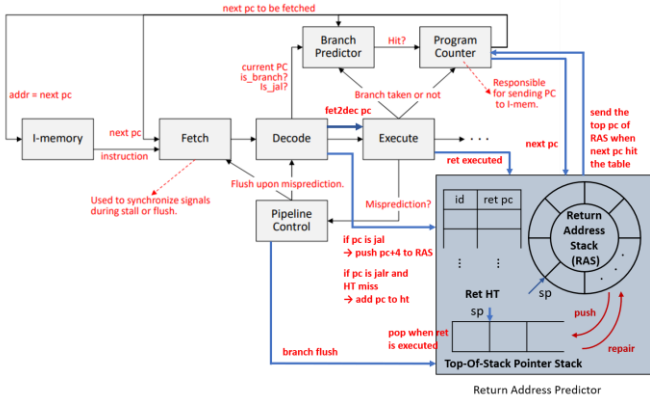

Fig. 4. Implementation of RAP with TOS

在儲存 TOS 的部分，我一樣用一個 stack 來存 RAS 在遇到 pop 的時候的 top stack pointer value，由於在 implement RAP 的時候，我從 RAS pop target address 並不是真的 pop，而是將 stack pointer 的值減一，因此 target address 的部分仍存於 RAS 的暫存器，所以值要我們可以回復 TOS pointer 的值，他所對應到的 target address 仍然不變，這也是我在上一部份將 RAS size 設為 4 的原因，來壁面 target address 太容易被覆蓋。

在實踐方面主要就是在 RAS pop 的時候，先將 TOS poiner 存到 stack，如果 execution stage 確定 ret 指令執行便將其從 stack pop 掉，同時，當 branch flush 的時候，將 RAS 的 stack pointer value 設成存於 stack top 的 TOS value。

B. Result of Implementing TOS Pointer Stack

TABLE V. THE PERFORMANCE OF IMPLEMENTING TOS (SIBLING ON)

TYPE	Iteraion/Sec	Total Cycle	Mispredict	HT Miss
RAP Off	101.198059	1540516618	X	X
RAP On	101.730150	1532478146	5122806	105
TOS On	101.737602	1532366184	5066828	105

TABLE VI. THE PERFORMANCE OF IMPLEMENTING TOS (SIBLING OFF)

TYPE	Iteraion/Sec	Total Cycle	Mispredict	HT Miss
RAP Off	100.718267	1548080490	X	X
RAP On	101.561194	1535265144	4255124	12531
TOS On	101.573985	1535072328	4158718	12531

TABLE V 討論了 RAP off，RAP (ret HT=256, RAS=4)，和啟動 TOS 的 performance 差別，可以發現，在啟用 TOS 之後，Iteration/Sec 上升，同時，mispredict 下降。而 TABLE VI 則是老師在上課提到可以關閉 compiler 對 sibling 的優化(-fno-optimize-sibling-calls)，在進行嘗試之後，可以發現在關閉優化後，performance 的上升更明顯，可能是因為關閉優化之後可以讓 RAP 的預測更準確，也可以看出，在加入考去 TOS 之後，確實可以提升 performance。

VI. CONCLUSION

在前半部分的分析中可以發現，在啟用 BPU 之後可以大幅度的提升 performance，同時可以看到 unconditional jump 在一定的 BHT size 之後可以達成幾乎 100 %的 hit，而針對不同的 Benchmark 程式碼，在節省電路面積的條件下也可以根據 Amdahl's Law 找到足夠大的 BHT size。

而後半部分則討論了針對 RAP 的設計，同時也在 RAP 中加入了紀錄 TOS pointer value 的 unit，而這樣的設計可以提升處理器的 performance，在 TABLE V 和 VI 中可以發現，雖然這樣的設計對 performance 的影響沒有很大，卻可以提升效能，推測這樣的設計在遇到 pipeline stage 較長的處理器可能影響會更顯著，因為 RAP 可以減少很多等待 execution stage 算出 target address 的 stall，而 TOS 可以減少 RAP mispredict 造成的 flush。

REFERENCES

[1] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark, "Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms," pp. 260-262.