HW#2 Report

賴御安, 110550168

***Abstract*—本文針對branch history table進行分析, 使用不同entry數來得到branch hit/miss的rate和iteration/sec。接著對於獲得的數據進行分析與討論。第二部分利用實作two level predictor (concate or gshare) 來進行分析, 看會對於預測的效能發生甚麼樣的改變。(*Abstract*)**

Keywords—branch predict; BPU; branch history table; BHT; one-level predictors; two-level predictors; gshare; concate; global history table; Aquila; verilog(key words)

I. Introduction

這次作業將利用hw1所學到分析電路cycle數的能力來獲得branch predict的hit/miss rate。在這次的分析中, 我嘗試調整branch history table的大小, 也就是對ENTRY_NUM進行調整, 另外也嘗試在關閉BPU的情況下去抓cycle數。在此篇中我也會套論hit/miss rate 和iteration/sec 之間的關係。第二部分則是實作two level predictor, 我總共實作了concate(將global history table跟pc的部分index concate作為index pattern history table)和gshare(將global history table跟pc的部分index 進行XOR作為index pattern history table), 並對得到的數值進行分析。

II. Analyze Branch Prediction

A. 分析*BPU*

我將分析用的電路放在bpu.v中, 透過is_cond_branch_i/ is_jal_i去判斷是否為branch, 再藉由將branch target addr 跟 dec_pc_i比較, 則可分出現在的branch的種類。對於branch miss 的判斷, 則是在前述的條件下加上沒有hit到BHT的情況時有branch_taken_i或是branch_misprediction_i。因為若是沒有hit 到BHT, 會在execute stage 才給定正確要執行的pc, 因此產生stall cycle, 而stall cycle才是影響cpu整體執行。misprediction則是因為執行到錯誤的指令, 因此要退回原先的狀態, 回復到預測前的狀態, stall cycle便是在這時候產生。

B. 修改*BPU error*

原先的branch_inst_tag是使用pc_i部分的bit讀出來的tag, 但是在execute stage時寫入則是使用dec_pc_i。因此原先的tag並不能作為we的判斷條件, 而是應該使用dec_pc_i讀出來的tag。而我則是透過在原本電路加上pipeline的方式向後傳兩個cycle(如過有stall cycle時則讓tag維持原本的值), 使得在execute stage能正確的比較tag值與dec_pc_i。這樣的好處是不需要對任何模組的I/O端進行修改。

C. 修改*BPU entry number*

對於修改BPU的entry number, 直接對ENTRY_NUM進行設定就可, NBITS會依照ENTRY_NUM對2取log, 得到read / write addrress的大小 (用來index BHT)。BPU_BHT的參數也會因為修改ENTRY_NUM後被設定成修改後的值。

要關閉 BPU predict 的功能需要註解掉aquila_config.vh中define ENABLE_BRANCH_PREDICTION的那行, 這樣在program_counter.v中就會啟用另一種給定下個pc的方式(位在第147-152行), 也就是要branch的pc會等到execute stage才給定。同時在分析branch miss (這時應該說因為branch而產生的stall cycle)的條件也要進行修改。因為主要是為了觀察發生stall cycle的情況, 因此我將設定為branch taken的時候, 在此情況才會有stall cycle, 而造成執行速度的降低。

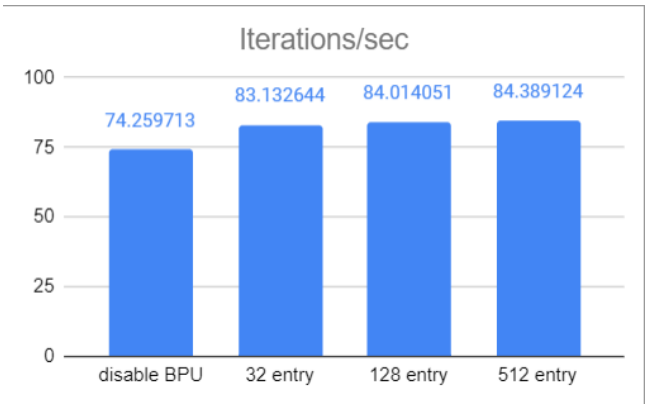
III. Analyze simple 2-bit predictor Result

對於分析BPU的performance, 我從branch hit/miss rate和Iterations/Sec的值這兩個地方下手。Branch miss rate越高, 則代表hit到BHT的次數較少, 因此則會產生stall cycle (對於disable BPU時則為branch造成stall cycle發生的次數)。一旦stall cycle的數量增加, Iterations/Sec則勢必會下降, 這就能很好的反應出performance的差別, 也可比較兩者之間的關聯性。

底下總共給出我有測試的4種設定, 分別為entry number為512、128、32及沒有使用BPU時的branch hit/miss rate和Iterations/Sec。會選擇512、128、32entry的原因是想讓差值能增大, 希望能更好的看出差距, 並討論這些數值所代表的意義。

D. 各項數據

disable BPU	miss	hit
conditional forward	0.1970171004	0.8029828996
conditional backward	0.8925118637	0.1074881363
unconditional jump	1	0
32 entry	miss	hit
conditional forward	0.1773382241	0.8226617759
conditional backward	0.2967731132	0.7032268868
unconditional jump	0.2443347528	0.7556652472
128 entry	miss	hit
conditional forward	0.1623261798	0.8376738202
conditional backward	0.2872852356	0.7127147644
unconditional jump	0.03616079829	0.9638392017
512 entry	miss	hit
conditional forward	0.1569423766	0.8430576234
conditional backward	0.2690706993	0.7309293007
unconditional jump	0.002873701101	0.9971262989



透過這些數據, 我發現在關閉BPU的預測功能後, 影較大的為conditional backward branch和unconditional jump。這個數據也十分合理, 因為pc在正常情況下會預設前一個pc加上4, 而如果需要做branch時就會產生stall cycle。且在branch的行為上, backward branch中會有一部分屬於迴圈最後向後跳的指令。當不對這種branch做預測時, 則會使得miss rate增加許多。Unconditional jump更不用說, 這種型態的branch會直接做branch, 當然不對此做預測則會讓miss rate來到最高。

在這些數據中, 在entry number增加時, hit rate會逐漸增加, miss rate逐漸降低, 表示我們產生的stall cycle數量有再減少。其中, 在32 entry到128 entry時 unconditional jump的miss rate減少的較為明顯, 這是因為對於此型態的branch需要做的預測只有taken, 會導致miss的情況應是當BHT中存放的位置被其他的指令覆蓋。因此當entry會大, 會被覆蓋掉的機率就會降低, 那預測的準度自然會上升。512 entry中unconditional jump的miss rate也減少許多。

接著討論啟用BPU時conditional branch的情況, 觀察表格可以發現對於entry number增加, hit/miss rate只有略為進步一些。我認為這是因為32個entry對於迴圈的branch predict已經足夠大, 表示迴圈中的branch指令已經不會覆蓋掉迴圈跳回去的指令在BHT的位置。至於因為entry增加而導致的miss rate減少(hit rate增加)則可看作是降低其他指令在BHT會被覆蓋掉的機率。

比較conditional和unconditional的branch可以發現, 在512bit時unconditional jump的miss rate已經降到非常低, 而conditional branch 則大約停在一個數值附近。這是因為對於unconditional jump, 只要辨識出來, 則只需要做taken預測就完成了, 而conditional branch 則會受到不同情況而影響到taken或是not taken, 因此對於條件的變化而影響的部分, BPU要做出正確預測的難度相當高(不排出可能有更好的預測方式)。另外在對迴圈中的預測中也勢必會產生預測失敗, 透過2-bit predictor可以將預測失敗的次數降為一次, 但仍然對於hit/miss rate會有影響。

另外, 我們可以發現iterations/sec在有無啟用BPU的差別很大。而當調整entry size時, iterations/sec也會有些差距, 但是, 當entry size從32到128時增加的幅度較大, 而從128到512時增加的幅度卻變少了。這可以從以前學到的Amdahl's Law來解釋此情況。在不斷提升branch prediction部分的效能時, 不能做branch prediction的部分會開始限制iterations/sec, 而且限制的幅度會越來越大。這時提高其他地方的效能會比持續增加BPU的entry size或預測效果來的有效。

IV. Two-level Branch Predictor來改進*BPU*的想法

目前我的想法為實作two-level的branch prediction unit中的gshare。且應該會以conditional branch主要觀察對象, 觀察是否透過gshare, 能夠提高現有的BPU預測的效果。

在觀察現有的BPU後, 我預計透過將distri_ram的data中dec_pc_i的部分進行修改, 利用額外的register存global BHT的資料與其進行XOR。而在讀資料時, 則在將對應的tag與pc_i進行XOR, 透過先前新加入的pipeline傳到exe stage與dec_pc_i比較。

關於實作global BHT的方式應該為利用額外的register存global branch history。利用簡單的shift概念應該能實作成功。

V. Global History Table 實作

在解釋Two-level Branch Predictor 實作前, 先解釋我對global history table的實作。GHT的大小將會根據不同的Two-level Branch Predictor設為不同的GBHT_SIZE, 這是因為concate的global history table是跟pc部分的index連載一起, 而gshare則是需要兩個部分bits樹一樣。

如果是concate, 則將entry的bit數量(NBIT)除以2, 若是無法平分時, 則會使pc的index多1 bit。用來index concate的pattern history table被pc跟的GBHT_SIZE平分的原因是希望可以將兩者的比重盡量靠近, 確保可以觀察出concate的預測效果, 而不是過於倚靠其中一部分。

對於gshare則是依照原本的定義將GBHT_SIZE與entry的bit數量(NBIT)設成一樣的bit數量。

對於更新global history table, 我將最新的branch記錄在bit 0, 並將原先的global history table 中全部向左移動一個bit, 並同時去除最舊的branch history。另外我有額外存一個用來restore的GHT(restore_global_BHT), 在更新global history table時更新一個相反的branch記錄, 而剩餘bit則和global history table作一樣的操作。當遇到misprediction時則會將restore的GHT改掉原先的global history table。這樣才能記錄到正確的branch history。

另外, 為了避免對於pattern的read與write不同步, 我在fetch stage與execute stage使用的global history table會相差兩個cycle, 處理方式與先前對於tag的操作是一樣的。這可以確保不會因為兩個靠近的branch出現read index與write index不同, (pc會是一樣的但是global history table可能不同)。

VI. Two-level Branch Predictor 實作

對於Two-level Branch Predictor 的實作, 本篇中實作concate(將global history table跟pc的部分index concate作為index pattern history table)和gshare(將global history table跟pc的部分index 進行XOR作為index pattern history table)。底下會分為兩個部分進行解釋。

*E. concate*實作

對於concate的實作, 我將global history table與pc的部分index合在一起作為pattern history table的index, 並寫入read_PHT和write_PHT。這兩個值則負責存取和寫入 history table和branch_likelihood。

*F. gshare*實作

對於gshare的實作, 我將global history table與pc的部分index進行XOR, 兩者的bit數量會是相同的, 並寫入

read_PHT和write_PHT, 用來負責去index history table和branch_likelihood。

VII. Analyze the Result for Global History Table

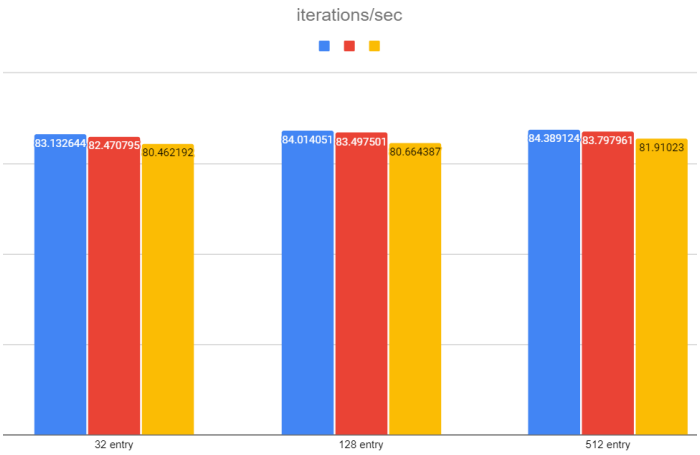


Fig. 1. 藍色是simple 2-bit branch predictor、紅色是concate、黃色是gshare

G. *iterations/sec* 數據分析

我先是將上一部分是用的參數套入concate和gshare進行測試, 上圖是我得到的數據。可以發現iterations/sec, 也就是效能的部分並未變得更好, 反而有下降的趨勢。根據我的推論, 我認為這是因為entry size 而限制了concate 和gshare的效能。對於同一個branch指令, simple 2-bit branch predictor只會佔據history table中的一個位置。而當開始參考global branch history後, 因為會將global branch history 與pc進行concate / XOR, 使得同一個branch指令在剛進入且global branch history尚未穩定時, 有可能會需要對多個history table進行寫入的動作, 覆蓋掉許多的entry。

透過觀察上圖的數據可以發現, 當entry數量增大時, 舉例:concate在32 entry與128 entry間、gshare 在128 entry與512 entry間, 預測的效能對比simple 2-bit的predictor在同樣範圍中增加許多。因此, 我將測試的entry number再次增加到2048 entry來觀察效能的差別。底下則為測試後獲得的資料。

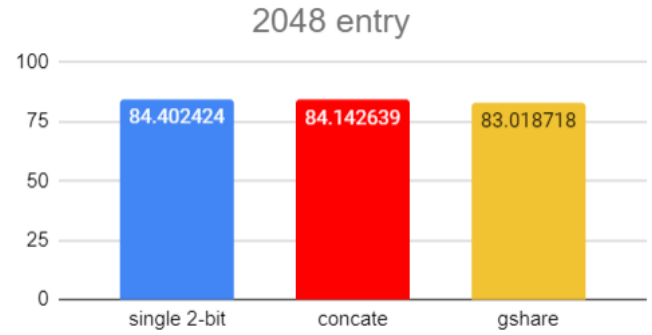


Fig. 2. 藍色是simple 2-bit branch predictor、紅色是concate、黃色是gshare

對比於512 entry的iterations/sec可以發現concate跟gshare的iterations/sec都有在成長, 而simple 2-bit branch predictor明顯開始出現瓶頸, 成長的幅度不大。但是, simple 2-bit branch predictor仍然還是有較好的效率, 根據前述, 可以推測是因為entry還未到concate / gshare適合的index。

H. *hit/miss rate* 數據分析

這部分的分析則只對三者iterations/sec較接近的case進行分析, 也就是2048entry。我認為這可以簡化分析, 看看三者對不同branch的效率差異。

single 2-bit			
rate		miss	hit
conditional forward		0.1569407085	0.8430592915
conditional backward		0.268630706	0.731369294
unconditional jump		0.000019419445	0.9999805806

concate			
rate		miss	hit
conditional forward		0.1550486046	0.8449513954
conditional backward		0.2990892487	0.7009107513
unconditional jump		0.04220580766	0.9577941923

gshare			
rate		miss	hit
conditional forward		0.1490302013	0.8509697987
conditional backward		0.3303262444	0.6696737556
unconditional jump		0.2206472048	0.7793527952

透過上面數據, 可以發現conditional forward的miss rate在使用concate和gshare時比single 2-bit來的效果好, 表示將global history table考慮進來後, 確實有提升部分預測的效果。而我認為conditional backward下降的原因則是當global history table較大時, 需要花費更多的branch才能穩定預測的值, 因此對於一個第一次遇到的迴圈就會產生更多不在entry裡的情況。

而三種branch預測中, unconditional jump是誤差最大的branch, 我認為這跟上述一樣, 在穩定情況前, 作出的預測都會錯誤, 且因為unconditional jump比較難預測什麼時候會發生, 若有透過不同函數呼叫同個branch, 則會覆寫原本的值, 便會發生錯誤預測。

VIII. 結論

對於BPU的實作, 我認為需要將能使用的entry size 納入重要的考量, 確保不會因為entry size而使得預測的效能不夠或是過於浪費, 才能在適當的entry size得到更好的預測效果, 提升CPU整體的效能。