HW1: Real-time Analysis of a HW-SW Platform

楊永琪, 109654020

***Abstract*—此為作業一報告, 嘗試使用 ILA 與撰寫 profiler 分析 CoreMark 跑在在 Aquila 與電腦上之間的差異。**

I. Introduction

這次的作業為撰寫 profiler 並使用 ILA 去對從電腦上跑出的5個 hotspot function : core_list_reverse, core_list_find, core_state_transition, matrix_mul_matrix_bitextract 及 crcu8 進行分析與比較。接下來會先介紹如何建立 profiler, 再針對 5 個 Hotspot function 在 Aquila 上執行的結果進行分析, 最後再針對電腦與 Acquila 上跑的結果進行比較。

II. Implementation

在 profiler 上搜集相關程式的執行 clock cycle 數、load/store 指令執行次數與在執行程式中 stall 的情形, 並將結過經 ILA 呈現。

其中, 執行的 clock cycle 數是藉由從 writeback stage 出來的 program counter (wbk_pc2csr wire), 在每個 clock 的 postive edge 去紀錄下來, 而 load 、store 指令則是藉由從 execute stage 所發送給 memory 的 write_enable, read_enable (exe_we 與 exe_re wire), 傳入 writeback 後, 經過一個 clock cycle 後傳出, 最後針對 stall 的研究則分為 stall_data_hazard 、stall_data_fetch 以及 stall_from_exe。

整體 CoreMark 執行 clock cycle 數則是取進入 main (PC=0x1088) 以及離開 main function (PC=0x1794) 之間所運行的 clock cycle 數。

III. Five Hotspot Functions

從筆電跑出的前5個Hotspot的結果依序為 core_list_reverse、core_list_find、core_state_transition、matrix_mul_matrix_bitxtract 與 crcu8, 接下來會針對在 Aquila 上跑出的結果進行分析。

TABLE I. Five Hotspot Function on Aquila

- main clock cycle count = 614876244

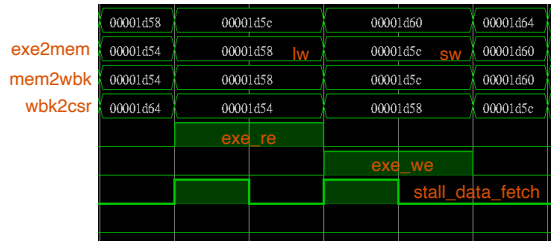
function	Five Hotspot Function on Aquila Analysis				
	total	*load/ store*	*data hazard*	*data_fetch stall*	*exe stall*
core_list_reverse	53546130	22215600	0	14810400	0
core_list_find	79614376	21199200	20825310	20949940	0
core_state_tranisi otn	113536720	16320480	4026880	1349320	0
matrix_mul_matri x_bitxtract	104248760	10986800	0	7458400	48308040
crcu8	68562102	0	0	0	0

TABLE II. Five Hotspot Function on Aquila Percentage

function	Five Hotspot Function on Aquila Percentage				
	total	*load/ store*	*data haard*	*data_fetch stall*	*exe stall*
core_list_reverse	8.71%	41.49%	0	27.66%	0
core_list_find	12.95%	26.63%	26.15%	26.31%	0
core_state_tranisi otn	18.46%	14.37%	3.55%	1.19%	0
matrix_mul_matri x_bitxtract	16.96%	10.54%	0	7.15%	43.34%
crcu8	11.15%	0	0	0	0

1. core_list_reverse

從 Table II 中可算出 core_list_reverse 占 CoreMark 總運行時間約 8.71% 以及 load/store 的占比為 41.49%, 細看此程式的組合語言, 確實發現他大部分的時間都在進行 sw 與 lw 的指令, 而在 Aquila 上進行 memory 的 read, write (exe_we | exe_re) 時並也與 ILA 的結果相符(如下圖), 皆會觸發一次 stall_data_fetch, 因此可以看到其有大量的 stall count。



2. core_list_find

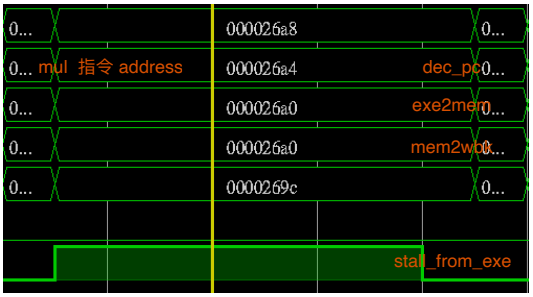
從表中可算出 core_list_find 占 CoreMark 總運行時間約 12.95%, 而其 load/store 的占比為 26.63%, 因為程式的組語中大部分的指令都有 load (lw, lh, lbu) 並接續 bltz, beqz, bne 等 branch 的指令去比較剛剛所 load 的值, 而這樣會產 load_use 的 data hazard, 其數據也與預期的相符, 可以從表中發現他 data hazard stall 的情況也大約佔 26.16% 左右。

3. core_state_transition

從表可算出 core_state_transition 占 CoreMark 總運行時間約 18.46% , load/store 的占比為 14.37%。此程式在他註解的地方標明是一個 State Machine, 在組語的部分可以看到有些 lw、sw 指令, 而從 source code 可以看到大部分的時間都在進行數值的比較以及將 state_tranisition counter 累加, 因此其 load/store 的占比與前兩個 function 相比少許多, alu 計算的時間佔比較多。

4. matrix_mul_matrix_bitextract

從表中可算出 matrix_mul_matrix_bitextract 占 CoreMark 總運行時間約 16.96%, 其中 load/store 佔比為 10.54%, 此程式主要在進行矩陣乘法, 以 alu 的運算為主, 並因為有大量的乘法, 因為 Aquila 乘法會算的比較久, 從下圖中可以發現當exe在執行 mul 指令時, 共會 stall 7 個 cycle, 而表中的 stall_from_exe 的 count 約佔程式的43.34%, 另外在讀取矩陣數值與寫入矩陣時也會用到一些 load/store 的指令。



5. crcu8

從表中可算出 crcu8 占 CoreMark 總運行時間約 11.15% 且沒有 load/store 指令, 看其 source code 會發現都在做數學運算, 沒有涉及到任何的 memory access, 也沒有使用乘除法, 為單純 alu 計算的 function。

IV. Comparison of Result on Computer

TABLE III. Hotspot Comparison

functions	Hotspot Result Comparison	
	Aquila(%)	*PC(%)*
core_list_reverse	8.71	28.39
core_list_find	12.95	25.26
core_state_transition	18.46	9.19
matrix_mul_matrix_bitextract	16.96	8.77
crcu8	11.15	7.86

根據 Table III 會發現在 Aquila 與在電腦上跑的結果完全不一樣, 在電腦上佔最多的反而在 Aquila 上佔得最少。

根據這張表, 在做 alu 的計算時, Aquila 會花比較多時間, 像是在做 matrix_mul_matrix_bitxtract 時, 約快要 1/2 的時間都是在 stall 等乘法的運算結果, 而電腦為可能因為有較多的資源, 針對此部分的計算有所優化所以對於他在跑 CoreMark 的時候影響不大。

而在單純做 load/store 指令時, Aquila 的效果比電腦的還要好, 可能是因為 Aquila 為 RISC-V 的微處理器, 處理記憶體存取的指令很有效率, 在執行 load/store 時每次只會 stall 1 個 cycle 就可以完成;而 CISC 在處理記憶體存取時的指令有許多種, 也比較複雜, 可能會等多個 cycle 以上, 導致其在電腦上跑的效能較差。

V. Summary

沒有想到第一份作業會這麼的困難, 在其中還是有些地方可以改進的, 像是在分析 stall cycle 時會有一點不準確, 因為我取的是在 writeback stage 裡的 pc 位於 function 區間裡的時候各個 stall signal 的值, 但由於進入 function 前後可能還是會有 stall cycle 產生, 所以可能會多算或少算到 stall 的值。

另外, 在討論 data fetch stall 時也卡了很久, 因為從 ILA 去看會發現每次有 load/store 時一定會 stall, 所以當時想的情況是 load/store count 的值應該要是他的兩倍, 但後來發現還可能會有 flush 或 exception 的情況發生, 排除此原因後去計算會發現有些 function 的 load/store 確實為兩倍, 但有些還是有一點差異。

這次作業雖然在 stall 方面的數據可能不是取那麼的精確, 但從中也可以看出根據不同 ISA 架構在運行程式時各方面的執行效率也可會有所差異, 覺得蠻有趣的。