HW1_Profiling CoreMark

黃柏竣, 110550123

***Abstract*—此作業用兩種方式對處理器跑分程式 CoreMark 進行分析，一種為軟體方式(gprof)，另一種為硬體方式(profiler in verilog)，並討論實作方式、兩者差別及未來優化可能性**

I. INTRODUCTION

Profiling 為一種技術，用來評估及量化程式性能，識別出 runtime 較長的函式，使在優化時，可以更好找到目標下手，以利程式效能的提升。而 CoreMark 為評估嵌入式系統和微處理機性能的跑分程式，所以此作業對 CoreMark 進行 profiling，分析後對程式更有概念，若要想辦法讓跑的分數更高時，更了解該如何起頭。

II. PROFILER

A. Gprof

GNU 工具，生成 call graph，顯示程式中各函式呼叫次數、執行時間等資訊，用來產生程式性能分析的資訊。

B. Profiler in verilog

在 aquila 處理器架構中，增加額外的電路，提供 profiling CoreMark 的功能，計算各函式跑 clock cycle 的數量，並透過 vivado ILA 得到存在 register 裡的值，計算並評估程式中五個函式的情況。

III. IMPLEMENTATION

A. Gprof and gprof2dot

利用 gcc/glibc/linux 編譯出讓 linux 本機可以跑的 CoreMark，並透過-pg 將使其執行時產生 profiling 相關資訊(gmon.out)，再用 gprof 得到 profile 文字檔，最後透過 gprof2dot 轉成圖片(Fig. 1)，並找出 Top 5 hotspots，作為硬體分析函式的參考。

B. Profiler in Verilog

從 gprof 得到的 5 hotspots 下手，去 coremark.map 看這五個函式的起始位置及結束位置，只要 PC 在那個函式的區段內，就判定為在執行那個函式，便在對應的 register 值+1。而 PC 的取值是從 writeback stage 拿，避免從前面 stage 拿，卻有因為 branch 等關係而沒實際執行到的情況發生。

總 clock cycle 的計算為判斷 PC 是否在 coremark 的 text section 位址範圍內，若是的話，就在對應 register 的值+1。

將 cycle 分類為 memory cycle 的方式為判斷來自 decode stage 的 write enable 及 read enable，一直接到 writeback stage，確保 PC 跟這些值是同 stage 的，若其中一者為 1 就是有對 memory 作讀寫，就判定為 memory cycle；而分為 stall cycle 的方式為判斷 execute stage 的 stall 訊號，一樣接到 writeback stage 確保同 stage，若為 1 就判定為 stall cycle。

而除了 memory、stall cycle 以外的就判斷為 computation cycle，就是函式總 cycle 數減掉 memory 及 stall cycle 數。

從 profiler 中取值的方式為 vivado 的 ILA，使用 set up debug，將想要看的值 mark 起來，並在透過 uart 傳入並執行 coremark.elf 後，紀錄 register 內的值。

IV. RESULTS

A. Gcc/Glibc/Linux

透過 gprof 及 gprof2dot 分析 coremark，得到 Fig. 1，並從圖中整理出 TABLE I.。.

TABLE I. TOP 5 HOTSPOTS IN COREMARK (UBUNTU HOST)

Rank	***Function name***	***Percentage***
1	core_list_find()	19.44%
2	core_state_transition()	18.86%
3	core_list_reverse()	15.79%
4	crcu8()	9.80%
5	matrix_mul_matrix_bitextract()	8.48%

B. Profiler in Verilog

透過 Vivado ILA 取得的值，比例的計算方式為各函式 clock cycles 除以總 clock cycles，而 memory 及 stall 的比例計算方式為該函式特定 cycles 數除以該函式 cycles 數，整理出的資訊如下面各表。

TABLE II. PROFILING COREMARK IN AQUILA FOR 5 HOTSPOTS

Rank	***Function name***	***Percentage***
1	core_state_transition()	18.48%
2	matrix_mul_matrix_bitextract()	16.95%
3	core_list_find()	12.95%
4	crcu8()	11.15%
5	core_list_reverse()	8.71%

TABLE III. COMPUTATION/MEMORY/STALL CYCLES PERCENTAGE

Function name	***Computation cycles Percentage***	***Memory cycles Percentage***	***Stall cycles Percentage***
core_state_transition()	64.39%	23.74%	11.87%
matrix_mul_matrix_bitextract()	78.55%	14.30%	7.15%
core_list_find()	21.06%	52.63%	26.31%
crcu8()	100%	0%	0%
core_list_reverse()	17.03%	55.31%	27.66%

V. DISCUSSION

A. *Difference between software and hardware*

因為軟體部分是 gcc/glibc/linux 編出來的 coremark，而硬體部分是 riscv-gcc/elibc 編出來的 coremark，執行的程式部分就有不同；再者，執行的架構也不一樣，軟體部分是在已經上市的處理器架構上面跑，且是 64 位元，而硬體部分則是在 32 位元的 aquila 上面跑，我想很多指令的執行狀況會在不同架構及環境下有所不同是很合理的。

但雖然排名有變，值看起來也都還是蠻大的(大於 8%)，再觀察軟體出來的結果，雖然在硬體部分沒有觀察所有函式，可是從上述推測，在硬體中的 Top 5 hotspots 也是這五個函式。

B. *Computation and Memory cycles*

從 computation cycle 及 memory cycle 中可看出函式對計算及 memory access 的作用時間比例，可以看到在整體 cycles 占比排名靠前的函式(前兩名)，都是 computation cycle 所花時間占比較多的，所以我認為，對此架構來說，對 computation 相關 operation 作優化相對於對 memory 相關 operation 作優化，會對 CoreMark 出來的結果有更好的效果，所以之後應該以加速 computation 相關操作為主。

crcu8()的部分，因為得到的值是 0，讓我有點驚訝，但去看了一下 objdump，看起來確實沒有 lw、sw 之類的指令，所以沒有 memory access 應該是沒錯的。

C. *Stall cycles*

若要處理 stall cycle 的問題，要先確認這些 stall 的主要起因，再對症下藥，可能因為 branch prediction 失誤、資料還在 load 等情況，使得 stall cycle 的產生，從 TABLE III. 中可看出 memory cycle 占比較大的，也會同時讓 stall cycle 占比提升，因此我猜想大多是 data dependency、cache miss 的問題，之後可以再針對這部分作處理。

D. *Future work*

從此次作業中，看出跑 coremark 這個跑分程式時，花比較多時間的函式，以及看出這些函式的各 cycle 佔據比例，因為在大多數情況下，大部分程式執行時間(80%)會聚集在小部分的 code(20%)，所以之後可以若要優化 Aquila 的 CoreMark 結果，就可以把心力放在找到的 Hotspot 上，像是 computation cycle 及 memory access 時常會遇到的 stall cycle，才能更有效地解決效能瓶頸。

APPENDIXES

Fig. 1. picture created by gprof2dot for CoreMark in ubuntu