HW3_Cache Optimization

黃柏竣, 110550123

***Abstract*—此作業先對 Aquila 原先運行之 cache 做分析，並在分析後，對 cache 的設定做修改調整，並評估比較這些方式在 CoreMark 的表現，藉這些實作更加了解 memory、cache 的運作及之間的關係。**

I. INTRODUCTION

先前的兩個作業都是將 instruction、data 放在 TCM 上，但因為 TCM 的成本高，實際上不太可能真的這樣子去實作處理器，所以此次作業多給了平常處理器常見的 cache 部分，並透過 CoreMark 這個跑分程式在執行時，分析 cache 的運作情形，像是 hit/miss、read/write 等情況。

除了原本的版本以外，也修改 cache 的 way 數量，以及 cache replacement policy，來進行測試比較，探討在 cache 方面的優化可能性。

II. IMPLEMENTATION

A. Setting of CoreMark for different number of ways in cache

此次作業所執行的 CoreMark 跑分設定，皆是在編譯時，將 Flags 的調為 "ITERATIONS=0 USE_CLOCK=1 TOTAL_DATA_SIZE=8000"。 在此基礎下，再對 D-cache 做改動及測試。

B. Background of all analysis in Homework 3

作業 3 測試的取得結果方式，是 CoreMark 在 Arty 板上跑完時，使用 Vivado ILA 取得各個 register 內的值，再去做計算及比較。

C. D-Cache hit/miss rate analysis

D-cache 的 hit/miss 分析是透過去數 processer 要從 D-cache 取資料的次數，以及在那其中又有 hit 到的值來判斷。Processer 要從 D-cache 取資料時，會送 p_strobe_i 到 D-cache controller，透過此來決定 processer 是否有 request 到 D-cache，若有，便在對應的 register 值增加 1(request count)；是否有 hit 則是在比較該 index 的所有 way 的 tag，如果有其中一個存的 tag 和 processer 的 request 對上，便將 cache_hit 設為 1，而我會在 p_strobe_i 及 cache_hit 時，在存 hit 數量的 register 加 1(hit_count)。

最後，hit rate 的計算就是 cache hit count register 的值除以 processer request count register 的值。

D. Average hit/miss and read/write latency cycles

Latency 的發生，是在 processer 對 D-cache 發 request，到 D-cache 拿到資料傳回給 processer 之間。根據我看到的 simulation 結果，cache hit 時，花一個 clock cycle 就有辦法將資料傳回給 processer，因此在此作業中沒有探討 hit latency cycle，將它認定為 1，也就是沒有 latency。

其他的部分，是去判斷是否在 p_strobe_i 之後，且在 p_ready_o 之前，這部分是用一個 register 做紀錄，processer 的 request 進來就設為 1，若資料 ready for processer，就設回 0。而各 cycles 的計算皆是在這兩者間，為了區分各個 latency cycles，還有加了判斷條件，總共分為 4 種 cycles，分別是 read hit、read miss、write hit、write miss，分別維持一個 register 給它們記錄數的數量，分類的方式則是去看 processer 送的 rw 訊號判斷讀或寫以及是否 cache hit，若情況有對應到下表，便在 register 內的值加 1。

	rw = 0	rw = 1
Hit	Read hit cycles	Write hit cycles
miss	Read miss cycles	Write miss cycles

關於這些 cycle 數平均值的計算，是有再做判斷呼叫 write 的次數，有多一個 register 記錄此值，若 request 來時，rw 為 1，便將值加 1(write count)，而各值的計算式如下所示。

$$average\ read\ latency\ cycles = \frac{(read\ hit\ cycles + read\ miss\ cycles)}{(request\ count - write\ count)}$$

$$average\ write\ latency\ cycles = \frac{(write\ hit\ cycles + write\ miss\ cycles)}{write\ count}$$

$$average\ miss\ latency\ cycle = \frac{(read\ miss\ cycles + write\ miss\ cycles)}{(request\ count - hit\ count)}$$

E. Cache replacement policy

此次作業中，加上原本的 FIFO，共使用五種 replacement policy，這邊會談我如何實作四種從網路上找到的 replacement policy。

第一種: LIFO，和 FIFO 很像，只是在每個 way 都填滿之後，就會只 replace 最後一個 way 的資料。

第二種: random，這部分是透過 clock 頻率很高，因此呼叫到 D-cache 時不一定會固定是 way_bits 的某個餘數的性質，模擬出類似 random 的狀況。維護一個 2 bits register 存 clock 除以 way_bits 的餘數，每個 clock 會更新值，若需要做 replace 時，就將值送到 victim_sel。

第三種: Most recently used (MRU)，是在每個 way 都填滿後，去看 hit 時的 hit index 為何者並記錄下來，需要 replace 時，將紀錄的值送到 victim_sel。

第四種: Pseudo least recently used (pLRU)，透過 tree 的形式去模擬 LRU，在 4 way 的情況下每個 index 僅需 3 個 bits，而非 LRU 的 6bits 及很多 state，因此選此方式實作。因為只要兩個 bits 就能決定一個 way，所以此方法是把被 access 的對應兩個 bits 設到另外一個方向，另一個 bit 維持不變，來讓最近被 access 到的 way 不容易再被 access 到。而這部份一樣是在有任一 way 被 access 時會對該 index 對應的這 3 個 bits 做上述的修改，並在 repalce 時，victim_sel 會根據這 3 個 bits，去選擇該 replace 的 way。

III. RESULTS

A. Hit rate, latency cycles and CoreMark score with different size of D-cache

在 aquila_config.vh 修改 D-cache 的大小後，進行測試，各值的計算如前面一個區段所示；以及進行 CoreMark 的跑分，從各結果整理出的資訊如下表 TABLE I.。.

TABLE I. AVERAGE LATENCY CYCLES, HIT RATE AND COREMARK SCORE WITH DIFFERENT SIZE IN 4 WAY FIFO D-CACHE

size of D-cache	*Hit rate*	*Average read latency*	*Average write latency*	*Average miss latency*	*CoreMark Score*
1KB	92.98%	4.824	2.580	50.25	3.974
2KB	98.44%	1.802	1.593	50.27	5.141
4KB	99.88%	1.057	1.088	50.32	5.570
8KB	99.99%	1.000	1.007	44.54	5.610

B. Hit rate, latency cycles and CoreMark score with different number of ways in D-cache

這部分皆是在 D-cache 8KB FIFO 的情況下測試，修改 D-cache 的 way 數後，進行 CoreMark 的跑分，從結果整理出的資訊如表 TABLE II.。.

TABLE II. AVERAGE LATENCY CYCLES, HIT RATE AND COREMARK SCORE WITH DIFFERENT WAYS IN 8KB FIFO D-CACHE

Number of way in D-cache	*Hit rate*	*Average read latency*	*Average write latency*	*Average miss latency*	*CoreMark Score*
1	99.75%	1.124	1.123	50.33	5.530
2	99.98%	1.006	1.010	49.75	5.607
4	99.99%	1.000	1.007	44.54	5.610
8	99.99%	1.000	1.006	43.31	5.610

C. Hit rate, latency cycles and CoreMark score with different cache replacement policy in D-cache

這部分皆是在 D-cache 2KB, 4-way 的情況下測試，得出的結果整理出的資訊如表 TABLE III.。.

TABLE III. LATENCY, HIT RATE AND COREMARK SCORE WITH DIFFERENT CACHE REPLACEMENT POLICIES IN 2KB AND 4WAY FIFO D-CACHE

Cache replacement policy	*Hit rate*	*Average read latency*	*Average write latency*	*Average miss latency*	*CoreMark Score*
FIFO	98.44%	1.802	1.593	50.27	5.141
LIFO	79.12%	12.606	4.590	50.32	2.519
Random	98.72%	1.613	1.727	50.33	5.218
MRU	79.12%	12.606	4.590	50.32	2.519
Pseudo LRU	98.72%	1.613	1.723	50.33	5.219

IV. DISCUSSION

A. Different size of D-cache

調整 D-cache 的 size 後，有看到說在 TOTAL_SIZE 為 8000 的情況下，基本上 8KB 是沒甚麼 miss 的，而調整到 4KB 的影響看起來也不大，可是若調到更低，miss rate 就開始有較高幅度地上升了，因此如果資源很吃緊的話，可以考慮將 D-cache 調整為 4KB。

B. Different number of ways in D-cache

這部分是在 8KB 情況下實作，可能會導致出來的數據不是那麼容易看到差距，但此部分還是能明顯看到 direct-mapped 跟其他之間的差異，也有看到基本上 way 數越高表現越佳，不過變更大之後，comparator 也要實作更多，吃掉較多的資源，但可能卻僅有小數點後好幾位的一點點進步，是否做越多 way 越好，還得謹慎評估。

C. Latency cycles

在 latency 的部分，有注意到基本上一次的 miss，就要花 40 至 50 左右的 cycles 去 memory 撈資料回來，跟 hit 比，有 4、50 倍的時間差距，可見 cache hit 的重要性，也實際見識到了 memory hierarchy 這兩層之間的差距。除此之外，也有注意到在 miss rate 在 1%以上時，read 的 latency 都比 write 的 latency 還要高，根據計算公式，可代表 read miss 是比 write miss 還要頻繁的，這部分的原因可能是因為剛好 read 都在 write 之前，或是其他原因，需要進一步探討，以利後續優化。

D. Different cache replacement policy

一開始在網路上看到 LIFO，覺得這不太可能會有效，但看到網路上有提供這方法，所以還是嘗試了一下，可是結果如同一開始所想，前三個不夠常被使用到，幾乎等於 1way 且容量還變四分之一的情況；而 MRU 也有同樣的狀況，因為前三個放入的資料幾乎沒被用，導致除了最後一個外，其它 way 根本沒機會被替換掉，也使它們的表現幾乎相同。而 random 表現出乎意料地好，和 pLRU 的表現差不多，不過和我的實作方法(照 clock)可能也有關係，不過根據它的表現看來，感覺 random 是個結構簡單且又較好實作的 cache replacement policy。