HW3 Cache Optimization

學號:0711282 邱頎霖

INTRODUCTION

一開始拿到為 FIFO, 4-way 的 cache 架構, 在計算小數點後 1000 時間為 1082 msec, 計算 5000 位數花費 37367 msec, 2-way 時執行時間在計算小數點後 1000 位數時為 1076 msec, 8-way 時執行時間在小數點後 5000 位數時為 37367 msec, 得到在計算小數點後 1000 位數時 2-way 較佳, 計算 5000 位數時 8-way 較佳的結果。分析後得到執行時間基本上含有 dirty bit 的 miss 成正向關係的結論。 接著以 LRU 替換 FIFO, 經過測試得到 2-way, 4-way, 8-way 在計算小數點後 1000 位數的時間分別為 1114, 1150,1142 msec, 計算小數點後5000 分別為37033, 36739, 37451 msec, 相比 FIFO, LRU 在計算小數點後 5000 位數時有更佳的執行時間與更低的 miss 次數。

ANALYZE

先使用老師提供的原始碼先進行 TCM 與 DRAM 的速度測試， 一開始就遇到速度與講義上有差異的情況，大概差了 10 msec， 例如 TCM 跑出來是 814 msec, 後來因為發生一些問題導致需要將工作區重新設定後再跑一次後速度就與講義上一致了, 而起始速度大致如 TABLE. 1. 所示， TCM 如預期的跑得最快, 可以作為此次作業 upper bound 。

接著根據作業要求將 cache size 固定為 2KB, 我們可以調整的部份大概為 cache set 內 block(line) 數量與 block size(line size), 同時因為cache size 固定, 因此一旦變動 set 內 block 數量也會相對應影響 set 數量, 如果對 set 內 block 數量進行改變測試, 結果如 TABLE. 1. 中所示, 可以發現 2-way 相比之下似乎速度快了一點, 而 4-way 與 8-way 在 PI 程式計算 1000 位數下速度近乎相同。

Type	TCM	2-way	4-way	8-way
time(msec)	804	1076	1082	1082

TABLE 1. execution time of PI program

在 cache 中較重要的議題是 hit 與 miss, 如果今天 cache 中 hit 次數越多, 我們可以減少需要到 memory 讀寫 data 等動作, 一旦越往底層的儲存裝置, 如 memory, disk 等等做讀寫的動作都是耗費相當大的時間代價的。 因此我對 hit, miss 做紀錄, 結果如 TABLE. 2. 所示 因為如果我們單純以 hit rate 下去排列, 計算出來的結果 2-way 會高於 4-way 與 8-way, 根據 TABLE. 1. 中的結果 2-way 的確在校能上優於 4-way 與 8-way 。 而 4-way 與 8-way 大致上速度差不多,後續會再討論兩者之間是否有差異性。

Type	write hit	write miss	read hit	read miss
2-way	0x1064da	0x1837	0x13a9e7	0x29090
4-way	0x0fccdb	0xb366	0x142f97	0x20ae0
8-way	0x0f8f73	0xf0ce	0x146d1b	0x1cd5c

TABLE 2. number of hit/miss in N-way Set Associative Cache

而為何 Hit 與 Miss 會是直接影響 cache 效能的直接因素, 可以從觀察 Aquila 內 D-cache controller 的 FSM 下手, 我們可以知道當最好的情況就是 IDLE 變化到 Analyze 時發生 hit, 而一旦發生 Miss 的情況, 我們還得考慮分別為當前為 dirty 與非 dirty 的兩個情況, 如果為 dirty 我們勢必還得花上時間將資料寫回 memory, 而且因為要對 memory 進行操作, 時間上的開銷一定不小, 根據使用 ILA 所觀察的結果, 如果要將內容寫回 memory(FSM 內須經過 Wb_to_mem 者), *p_strobe_i* 與*p_ready_o* 之間的 latency 數多達約 51 cycle; 如果不需要將 data 寫回 memory(FSM 只須經過 Rb_from_mem 者) 只需要從 memory 讀取到 cache 的話, latency 數約為 31 cycle。對此我分別紀錄單獨經過 Wb_to_mem 與 Rb_from_mem 的數據, 數據如 TABLE. 3. 表示, 這邊表格內數字為十進位制表示。 可以發現單獨經過 Rb_from_mem 的次數僅為 128 次, 而因為 dirty bit 導致須花較多時間的寫回記憶體的 miss 次數為 4-way 最多, 8-way 次多, 2-way 的次數相比上述兩者有著顯著的減少，也因此在效能上也有顯著的不同。 我們可以總結在執行時間上, 影響最大者為含有 dirty bit 的 Miss, 其次為不含有 dirty bit 的 Miss, 但是因為後者與前者相比少上許多, 因此在後續討論時我們可以先大致以含有 dirty bit 的 Miss 作為基準, 如果此項數據較大我們基本上可以推論說他需要較多的執行時間。

	Miss_With_dirty	Miss_Without_dirty
2-way	174327	128
4-way	179782	128
8-way	179754	128

TABLE 3. number of miss with dirty bit/ without dirty bit

從 TABLE. 3. 還可以觀察到一件事情為 8-way 似乎有一定的潛力可以比 4-way 有更好的效能, 儘管在 PI 程式計算小數點後 1000 位的情況下兩者執行時間大致相同, 但是如果要探討更多位數的話也許有不一樣的效果, 因此我多測試了執行 5000 次的情況, 結果如 TABLE. 4. 顯示, 在計算小數點後 5000 位數的情況下 8-way 較 4-way 還快了一點點, 另外 2-way 在計算 5000 位的情況下明顯變慢了, 有趣的是當我去紀錄三者分別含有 dirty bit 的 miss 與不含 dirty bit 的 miss 次數時, 三者結果跑出一模一樣的數據但是 8-way 跑得比較快。

	time(msec)	Miss_With_dirty	Miss_Without_dirty
2-way	37367	9858214	128
4-way	37367	9858214	128
8-way	37366	9858214	128

TABLE 4.

CACHE REPLACEMENT POLICIES

除了 block(line) size, cache 內 block 數量可以做調整, 還可以做調整的為 cache 替換的演算法, 較常見的為 FIFO, LRU, LFU 等, 原始程式碼中使用 FIFO, FIFO 簡單易實做, 但效果如何要視當下 cache 內的 pattern 後才知道。 如果 cache 內有一定的規律或是 pattern, 也許我們可以針對此規律選擇相對應較佳的演算法。

我使用 ILA 試圖去分析執行時每次需要 hit, miss 時的 line index 與 tag, 剛開始執行沒多久預期應該會是先有大量的 miss 填滿我們 cache 的內容, 才會開始有 hit 出現。 實際上則是先填入 cache 後馬上過沒多久就使用到 cache 內的內容, 且因為我有分開紀錄 write, read 的 hit, miss, 看到前期都被 write hit/miss 給佔滿, 完全沒有 read 的部份, 以 2-way LRU 為例一直到 write hit 與 write miss 分別累積到 0x7b8c 與 0x841 後才出現第一筆 read hit, 接著 read 與 write 則是分別互相交錯出現, 但以結論來說我沒有發現什麼特別的 pattern 或是規律, 因此我以前面提過得其他兩個 Cache replacement policies: LRU, LFU 的特性做選擇, 考量到 LFU 需要紀錄出現過的頻率, 並且 miss 時需要找出 victim 時也許會花上不少時間, 儲存頻率時空間也可能不夠的, 因此我選擇使用 LRU 試試看是否可以達到更好的效能, 儘管 LRU 會用上類似 timestamp 的概念去紀錄上次使用的時間, 但其實我們可以稍微簡化他, 單獨去維護一個 List, List 前面就是每次選擇的 victim, 發生 Hit 時就搬到 List 最後面, 以這樣實現的時間複雜度而言相比原本 FIFO 並不會花上多少額外時間。

LRU IMPLEMENTATION

在使用 LRU 後 PI 程式計算小數點後 1000 位數結果如 TABLE. 5. 所示, 可以看到使用 LRU 後 2-way 仍然是跑最快者, 速度與 FIFO 相比又再慢了一點。 但是如果以 PI 程式計算小數點後 5000 位數的話結果將會改變, 結果如 TABLE. 6. 所示, 可以看到如果要計算小數點後 5000 位數的話又會變成 4-way 變成三者之中最快的, 且使用 LRU 的 2-way 與 4-way 都比使用 FIFO 時還快了。

n-way	time	wHit	wMiss	rHit	rMiss
2-way	1114	0xfae1a	0xd227	0x13e9ee	0x25089
4-way	1150	0xf5be2	0x1245f	0x13c8ac	0x271cb
8-way	1142	0xf7de3	0x1025e	0x13c146	0x27931

TABLE. 5. Execution time of LRU (NDIGITS=1000)

n-way	time	wHit	wMiss	rHit	rMiss
2-way	37033	0x10065f9	0x32f9f7	0x14aa456	0x92f2e6
4-way	36739	0x100f4f5	0x326afb	0x14ce5c8	0x9022f8
8-way	37451	0x1001ccd	0x334323	0x146c83a	0x9718ae

TABLE. 6. Execution time of LRU (NDIGITS=5000)

根據前面所敘述, 在 cache 中的快慢應該要與有 dirty bit 的 miss 數量呈現正向關係, 因此我們可以在換成使用 LRU 後驗證此項推論是否依然正確, 首先前面有講到不含 dirty bit 的數量應該會比含有 dirty bit 的 Miss 數量少上許多, 這點到了 LRU 依然保有一樣性質。 在 LRU 中含有 dirty bit 的 Miss 一旦發生所產生的 latency 約為 50 clock cycle, 而含有 dirty bit 的 Miss 發生的話產生的 latency 約為 30 cycle, 到此基本上與 FIFO 大致上相同, 原因是在改寫成使用 LRU 時基本上相比 FIFO 我沒有花上額外許多的時間找出 Victim, 基本上就是維護一個 List 並且每次將最前頭的作為 victim, 一旦讀過或是寫過後就放到最後。 因此在時間複雜度上基本上與 FIFO 不會差上太多, 但是如果改使用 LFU 或是其他演算法的話可能都需要再額外上時間去找出 Victim, 也許他們可以降低 Miss 次數但就是得付出相對應的時間上的代價。 TABLE. 7. 為計算小數點後 1000 位數的結果, 可以看到與 TABLE 3. 使用 FIFO 時相比, LRU 有更多有 dirty bit 的 miss 數量, 也因此在執行時間上 LRU 需要更多的計算時間。 TABLE. 8. 為計算小數點後 5000 位數的結果, 可以看到與 TABLE 4. 使用FIFO 時相比 2-way 與 4-way 的 LRU 有更少有 dirty bit 的 miss 數量, 也因此在計算時間上他們有相比 FIFO 有更少的執行時間。

n-way	Miss_With_dirty	Miss_Without_dirty
2-way	205358	130
4-way	234917	133
8-way	228097	142

TABLE. 7. Number of miss with/without dirty bit (NDIGITS=1000)

n-way	Miss_With_dirty	Miss_Without_dirty
2-way	9630438	128
4-way	9446006	130
8-way	9902122	132

TABLE. 8. Number of miss with/without dirty bit (NDIGITS=5000)

SUMMARY

從一開始拿到為 FIFO, 4-way 的 cache 架構, 在計算小數點後 1000 時間為 1082 msec, 計算 5000 位數花費 37367 msec, 當改為 2-way 時執行時間在計算小數點後 1000 位數時為 1076 msec, 小數點後 5000 位數時為 37367 msec, 改為 8-way 時執行時間在計算小數點後 1000 位數時為 1082 msec, 小數點後 5000 位數時為 37367 msec, 得到在計算小數點後 1000 位數時 2-way 較佳, 計算 5000 位數時 8-way 較佳的結果。分析後知道含有 dirty bit 的 miss latency 約為 50 clock cycle, 不含有 dirty bit 的 miss 約為 30 clock cycle, 且含有 dirty bit 的 miss數量遠大於不含有 dirty bit 的 miss, 因此執行時間基本上含有 dirty bit 的 miss 成正向關係。 接著試圖去分析了執行 Pi 程式時 cache 內是否具有一定的 pattern, 雖然沒有找出相關的 pattern 但也說明了如何選擇 Cache replacement policies, 最後選擇 LRU 替換 FIFO, 經過測試得到 2-way, 4-way, 8-way 在計算小數點後 1000 位數的時間分別為 1114, 1150,1142 msec, 計算小數點後5000 分別為37033, 36739, 37451 msec, 相比 FIFO, LRU 在計算小數點後 5000 位數時有更佳的執行時間與更低的 miss 次數。

這次作業中遇到蠻多神奇的事情, 像是最前面提到的執行時間多跑幾次會有不同的情況, 還有有時 LUT 不夠導致跳出錯誤, 但是前幾次合成更大的 LUT 都不會跳出錯誤, 比較可惜的是時間上不夠, 沒有辦法去調整 block(line) size 大小, 否則根據以前學計算機組織的印象, block size 也會造成影響。 而且是第一次使用 Verilog 將 Cache replacement policies 給實做出來, 覺得頗為新鮮, 之前都是用 C++ 模擬過 LRU, LFU, 搭配 C++ 的 STL 要實做出來真的不難, 還可以跑得挺有效率的, 另外 FIFO 真的突破我對他的認知, 以前覺得 FIFO 就是最簡單最沒人會選擇的, 直到這次作業好幾次跑出來的時間都比 FIFO 差才敢受到他的強大。