HW#3 Report

賴御安, 110550168

***Abstract*—本文針對data cache進行分析, 當memory request時cache所產生的read/write latency及miss/hit所產生的latency。接著並嘗試調整cache way和cache replacement policy, 進而討論所得到的結果。(*Abstract*)**

Keywords—data cache; cache; FIFO; LRU; LFU; cache way; Aquila; verilog; (key words)

I. Introduction

在這次的作業中, 我將CoreMark的data size設為4000, 利用抓去特定訊號及所在的state去判斷memory request是read或是write, 也能判斷cache hit / miss, 並藉由不同條件下的flag獲得相對應的latency, 也就是個別的cycle數。

另外, 本次作業中我嘗試調整cache way的數量, 分別是2、4、8, 並配合不同的replacement policy, 分別是預設的FIFO、LRU、LFU。最後, 透過兩者搭配所獲得的數值進一步做出分析與討論。

II. counter電路實作

這次需要獲得的數據可以分為兩類, 一類為cache的hit與miss, 另外一類則是read / write hit / miss 的latency。底下分別敘述兩種的實作方式。

A. 加入 *cache hit/miss* 的 *counter*

我判斷cache hit / miss rate的方式是在Analysis state時去看cache_hit這個訊號。這麼做的是因為在Analysis這個state只會有一個clock cycle, 所以我便能在這個cycle取得hit與miss的次數。這也符合在一次的memory request中只會有一次的cache hit或是miss。我同時也紀錄read / write的hit / miss次數, 再透過下個部份得到的latency來算出average。

B. 加入 *read / write hit / miss* 的 *latency counter*

這次作業中我計算latency的方式是透過reg作為flag來判斷counter是否要增加。我先在Idle state時判斷接下來的flag形式。當p_strobe_i為1時, 透過p_rw_i來判斷memory request是read還是write, 並將對應的flag (read_latency_flag或write_latency_flag) 用non-blocking的方式賦值。這樣在Analysis state時便能使用這個flag來使對應的counter增加。另外特別提到, 判斷memory request是read還是write時使用p_rw_i的原因是因為我在的state是Idle, 應該要使用p_rw_i而非rw。

而接著計算miss latency的flag則是在Analysis state時才做更新, 在前述的flag啟動時, 同時p_is_amo_i跟cache_hit都為0的時候, 將對應的miss latency flag用non-blocking的方式賦值。因為是在Analysis state才用non-blocking的方式給定值, 因此計算miss latency的counter在Analysis state不會增加, 但是miss latency仍然需要包含這個cycle, 所以我的解決方式是在分析數值時將miss latency的值加上miss的次數並得到真正的miss latency。

III. D-cache modification

對於這次作業的另一部份則是對於data cache進行修改, 我在本文中總共對cache way和replacement policy進行修改, 將cache way的數量設定為2、4、8, 另外也嘗試了3種replacement policy, 分別為FIFO、LRU、LFU。底下會分別敘述我所做的修改方式。

C. Cache Way

對於cache way的修改, 除了修改localparam的N_WAYS之外, 還需要對assign way_hit的地方進行修改, 根據增加或減少的cache way數量來調整, 並調整cache_hit。另外, 也要根據增加或減少的數量來調整case使用的bit數量以及hit_index的範圍。

D. Replacement Policy

我總共多實作了兩個replacement policy, 1. LRU (Least Recent Use) 和 2. LFU (Least Frequency Use)。根據我分析原先的FIFO policy, 發現需要修改的地方為FIFO_cnt 跟victim_sel, 因此我從這個部份下手。底下會分別敘述實作的細節。

a) LRU

我新增一個LRU_cnt, 大小為N_LINES*N_WAYS個WAY_BITS大小的變數, 用line_index來取得第幾個line, 這個部份是跟FIFO是一樣的。不一樣的地方在於多加N_WAYS的部份, 這幫我存著各個line中不同way最近使用的情況。放在0的是最久沒用的, 3則是最近一次用的。因此victim_sel的選取會固定為LRU_cnt[line_index][0]。

對於保持LRU_cnt的狀態, 我在最開始將各個line N_WAYS的部份初始化為0~N_WAYS-1, 接著當在Analysis state時cache_hit, 則進行重排, 將hit到的way移到LRU_cnt[idx][3], 其餘則向前移動一格。當cache not hit時, 因為我不會對LRU_cnt的值進行修改, 因此victim_sel會讀到這個line中最久沒被使用的way 的number 並對這個way進行替換, 最後在RdfromMemFinish state將這個way重排, 移到LRU_cnt[idx][3], 剩下的向前移動。

b) LFU

LFU的實作則是建立大小為N_LINES*N_WAYS的counter, 用來存各個line的各個way使用的次數。當hit到對應的line中的way時, counter就會增加。而選取victim_sel的方式是用一個for迴圈不斷紀錄在line_index最少被使用的way的index, 並存在least_use中, 當沒有hit到的時候便選取least_use。而各個line的各個way使用的次數會在沒有hit到的時候歸零重新計算。

IV. 數據分析

在這次的實驗中, 我的data cache大小都設定為2KB。底下我會分別對三種replacement policy在不同的cache way下所得到的cache hit/miss 以及average latency進行討論, 並最後再做出統整。

E. FIFO

	FIFO 8way 2K	FIFO 4way 2K	FIFO 2way 2K
read hit	72,147,190	72,143,964	72,061,100
read miss	162,290	165,516	248,381
write hit	18,660,545	18,664,499	18,664,204
write miss	37,897	33,943	34,239
cache hit	90,807,735	90,808,463	90,725,304
cache miss	200,187	199,459	282,620

Fig. 1. cache 和read/write request的hit/miss的次數

read avg miss lat	50.31959455	50.3484497	50.40493838
write avg miss lat	50.16127926	50.15838317	50.27851281
avg cache miss lat	50.2896242	50.31610506	50.38962211

Fig. 2. average miss latency

由上圖可發現, read miss次數在way增加時會減少, 我推測這是因為way的數量變多, 因此cache在寫入後影響的範圍比較小, 所以在連續的read request搭配一兩個write request便能有更高的hit rate。而write miss的話, 我認為較沒有固定的規律。

F. LFU

	LFU 8way 2K	LFU 4way 2K	LFU 2way 2K
read hit	71,797,607	71,987,674	72,089,576
read miss	511,874	321,807	219,904
write hit	18,672,498	18,675,834	18,662,393
write miss	25,945	22,609	36,049
cache hit	90,470,105	90,663,508	90,751,969
cache miss	537,819	344,416	255,953

Fig. 3. cache 和read/write request的hit/miss的次數

read avg miss lat	50.32496083	50.33767134	50.32383222
write avg miss lat	50.28086336	50.23393339	50.24028406
avg cache miss lat	50.32283352	50.33086152	50.31206511

Fig. 4. average miss latency

由上圖可以發現, read miss的次數相當高。我認為這個是由於我實作的LFU的特性, 當這個line中有一個way被更新時就會將這個line的counter歸零。我在找尋victim_sel時會先假設第0個way是最少被使用的, 如果這時在其他way也有沒被使用的話, 就不會被選擇要更新。這可能是導致miss rate較高的原因。

G. LRU

	LRU 8way 2K	LRU 4way 2K	LRU 2way 2K
read hit	72,127,815	72,005,370	71,978,842
read miss	181,665	304,111	330,639
write hit	18,660,805	18,627,455	18,648,085
write miss	37,637	70,988	50,358
cache hit	90,788,620	90,632,825	90,626,927
cache miss	219,302	375,099	380,997

Fig. 5. cache 和read/write request的hit/miss的次數

read avg miss lat	50.36410976	50.31941955	50.40800087
write avg miss lat	50.26019608	50.2560151	50.33537869
avg cache miss lat	50.34627591	50.30742017	50.39840209

Fig. 6. average miss latency

透過LRU replacement policy得到的數據與FIFO有高度的相似性, 都是在read miss的way數量增加時會減少, 而write miss則是相對沒有固定規律。我認為數據這樣呈現的原因與FIFO相似。這樣推測的原因是由於LRU與FIFO在選擇取代的way時都是取最舊的way(可以將LRU看作是把最近使用的way當作最新)。

H. 綜合討論

透過看average miss latency, 就可以發現如果沒有hit到cache, 會需要大約50個cycle來從memory讀取資料到cache, 由此也呼應老師上課所述, cache對比於onchip memory來說會降低效能。

透過觀察這三個replacement policy所得到的數據, 可以發現並不是讓replacement policy複雜就能提升效能, 而是要找到對於現在情況最適合的。我認為LFU的效能比較差的原因是由於在同一個line中可能會有很多個way沒有被使用, 這時可能會有很久沒有使用的way因為在較後面所以沒有被更新到。因此, 會把比較久以前進來的cache data優先丟到的, 也就是FIFO跟LRU才是比較適當的replacement policy。

而針對FIFO跟LRU這兩者效能上的差別, 我會歸咎於CoreMark的程式較為適合FIFO而非LRU。但是, 我認為LRU在其他環境下是有可能跑得比FIFO還要好的。