Computer Architecture

Final Project Report

學號：102062801
姓名：孫勤昱

Outline

Algorithm description

這次的模擬 cache 的作業中，最主要的兩個 algorithms 分別為:
(1) Replacement 機制中，用來挑受害者的 LRU 演算法，和
(2) 用來降低 miss rate 的 Zero-Cost algorithm

ALU algorithm

ALU 演算法，我使用了 Stack 和 Linklist 兩種方法來模擬近似 LRU 的方法。主要根據 input 的 data，我把 ALU 分成 1，2，和 N associativity 三種 cases (associativity)。首先，我們先討論 1 和 2 associativity。當進入這個 case，我利用 stack 來設計 LRU。若是 associativity 為 1，他會執行 directmap (access_set_directmap)。而 2，他會進入 twoWayAssociate。它先取出檔案的 ref list，並用 ref 跟算出的 indexMask 去做 and 運算。運算完畢後就會得到這個 reference 應該索引第幾個 set 了。當以上執行完畢後，開始進入 LRU 演算法 (access_set_lru2way)。演算法中，首先它會先搜尋看 set 裡面有沒有存在 ref。如果 i 不是第 0 格就把它放到第 0 格 (就是 stack 的方式，會塞到最頂端)。換而言之，底端的就是要準備換掉的。另一方面，如果沒找到就有兩個情況：1) 如果 cnt 是 0 表示 set 裡面沒東西，可以將資料直接放進去，和 2) 如果有東西，表示有 1 個或 2 個(但找不到)，這時直接把原本最前面的塞到底端，把新的放到頂端。

但 associativity 超過 2 的時候，因為找不到更快的插入法，所以決定用 Linkedlist 來實現。當進入到 N associativity 的情況下，我讓程式碼產生 ListSet 陣列，陣列每個元素都是一個 ListSet。然後做一樣的事情，取出 ref，index，indexValue，並執行 access_listset_lru。此時，access_list_set_lru 會看說 set cnt 是不是 0。若是 0 就是空的，我們可以把資料直接放進去。反之，他會搜尋。若搜尋到的話會把找到的節點放到最前端(程式中用 listset_move_to_front)。如果都沒找到，但 listset 還有空位就插到最頂端。若沒空位，他會直接選擇最尾巴的，把最尾巴的那個的 ref 改為新來的 ref，最後用 listset_move_to_front 移動到最前端。

Zero Cost algorithm

而在 Zero-Cost 演算法，我大致將其演算法細分了三個 steps，分別敘述如下：

在第一個 step，主要是要算出 Q 值(範圍介於 0 和 1)。以圖(一)為例，我們將 A_5 內 0 的個數(= 8)與 1 的個數(= 2)。因為要保持 Q 值介於 0 和 1 之間，我們要將大(小)的數放分母(分子)。因此，我們可以得到，$Q_5 = \frac{2}{8} = \frac{1}{4}$。經過以上的算法，我們可以將所有的 bit(扣除 offset)都算出 Q 值。

A_5	A_4	A_3	A_2	A_1	A_0
0	1	1	0	1	1
0	0	1	1	0	0
0	0	0	1	1	0
0	1	0	0	1	1
1	0	1	0	1	1
0	0	0	1	0	0
0	1	1	1	0	0
0	0	0	0	1	1
0	0	1	0	1	1
1	0	0	1	0	0

圖(一) Example for calculating Q

在第二個 step，主要是要算出 $C_{i,j}$ 值(任兩個 bit pair 算 i 與 j 的相似度)。以圖(二)中 A_4 與 A_5 為例，我們要看 A_4 與 A_5 中一樣 bit 的次數(= 5)和不一樣 bit 的次數(= 5)。因此，我們可以得到 $C_{4,5} = \frac{5}{5} = 1 = C_{5,4}$。並藉由 step 2 的算法，每兩兩算出 C 值來得到完整的圖(三)。

	A_5	A_4	A_3	A_2	A_1	A_0
不一樣 →	0	1	1	0	1	1
一樣 →	0	0	1	1	0	0
一樣 →	0	0	0	1	1	0
不一樣 →	0	1	0	0	1	1
不一樣 →	1	0	1	0	1	1
一樣 →	0	0	0	1	0	0
不一樣 →	0	1	1	1	0	0
一樣 →	0	0	0	0	1	1
一樣 →	0	0	1	0	1	1
不一樣 →	1	0	0	1	0	0

圖(二) Example for calculating $C_{i,j}$

	A_5	A_4	A_3	A_2	A_1	A_0
A_5	0	1	1	1	2/3	1
A_4	1	0	2/3	2/3	1	2/3
A_3	1	2/3	0	2/3	1	2/3
A_2	1	2/3	2/3	0	1/9	0
A_1	2/3	1	1	1/9	0	1/9
A_0	1	2/3	2/3	0	1/9	0

圖(三) Results of $C_{i,j}$

在第三個 step 中，我們要將前面提及的 step1 和 2 算出的結果來 training。透過論文中提及的 algorithm，我們每經過一回合的 training，就可以得到一個最佳的 bit 當作 index 來降低 miss rate。以圖(四)中的 algorithm 為例，一開始我們初始 M=6，S 集合為空。i 從 0 跑到 5。A_{best} 等於 Q_5 到 Q_0 最大值(順便檢查有沒有在 S 集合內。若找到 A_{best} (假設 A_0)，就把這個 A_0 放入 S 集合(避免下次再次比 A_0)。接著，我們依順序算 $Q_0 = Q_0 * C_{best,0}$，$Q_1 = Q_1 * C_{best,1}$，…，$Q_5 = Q_5 * C_{best,1}$。第一回合算完後結束，我們得到 $A_{best} = A_0$ 並得到更新過的 Q_5 到 Q_0(如圖五)。下一回合用更新過的 Q 值來找最大，repeat 演算法。(第二回合 Q_3 最大，$A_{best}=A_3$)

Algorithm 1. Computes Near-optimal Index Ordering

```
Input: M {the size of the address space}
Input: Q_{M-1}...Q_0 and C_{M-1,M-1}...C_{0,0} {quality and correlation measures}
S ← ∅
for i ← 0 to M − 1 do
    A_best ← max(Q_{M-1}...Q_0)|A_best ∉ S
    S ← S ∪ A_best
    for j ← 0 to M − 1 do
        Q_j ← Q_j * C_{best,j}
    end for
    print A_best
end for
```

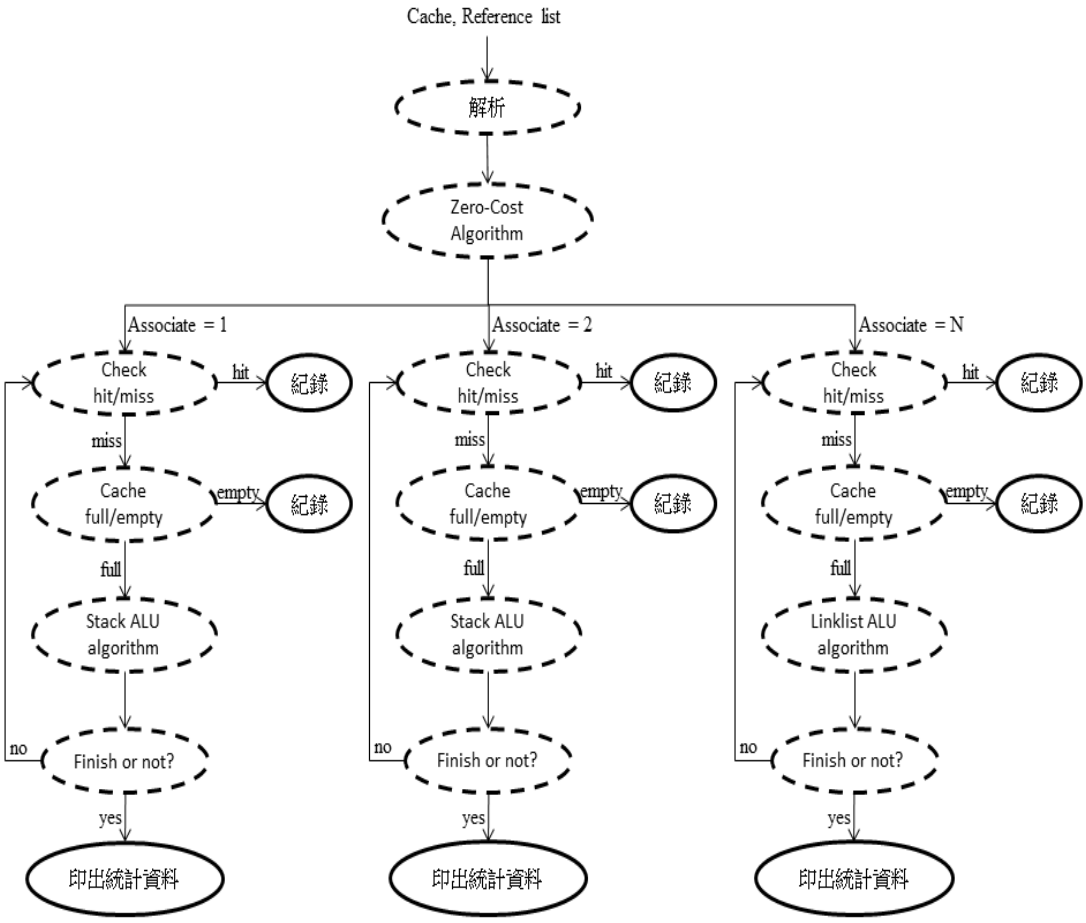
圖(四) Training algorithm

Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
1/4*1 = 1/4	3/7*2/3 = 6/21	1*2/3 = 2/3	1*0 = 0	2/3*1/9 = 2/27	1*0 = 0

圖(五) The updated values of Q after finishing the first round

Flow-chart of algorithm

圖(六)，為程式碼的 flow-chart 圖。首先透過解析，可以將助教的 input 檔案轉換成能用的數值。透過解析出來的值，我們接續執行 Zero cost algorithm。其 Zero cost 的詳細演算過程，在上一章節之第二點有討論。接著藉由解析出來的資料，會分成以下三種 cases。Associate 為 1(相當於 dirct-map)，Associate 為 2，和 Associate 為 N。這三種 cases，大致上會做一樣的事情。檢查 Hit 還是 miss，Cache 是否滿的，與印出統計資料。唯一不同處為 ALU 的實現方法，前兩種為 Stack 實作，最後一種 case 為 linklist 的方法來實現。



圖(六) Flow-chart of algorithm

Summary of result

Cache / Reference	A	B	C	D
DataReference_n_comp	545	168	561	168
DataReference_n_real	312	199	239	101
DataReference_real_up	47	35	41	35
InstReference_iir_one	1309	882	1323	872
InstReference_lms	2052	992	2196	944
InstReference_matrix	4044	997	4821	949
InstReference_n_comp	4441	1057	4557	1009
randcase1	3000	3000	3000	3000

Discussion

這次的 project，透過老師在課堂中詳細介紹了 cache 的運作原理與 LRU 演算法，我實作了一個 cache 的運作。基本上這些功能 coding 時候沒遇到太大的問題，反而為了要降低 miss rate 並要在一分鐘內跑完程式，我花了很多時間研究。我自己寫了一個暴力法來測最低 miss rate，但超過了助教規定的執行時間太多了。另一方面，我也有上網看看有沒有其他論文改進助教提供的 zero cost algorithm 的論文，但我並沒有看到新的方法來改進 zero cost algorithm 的論文。因次，我決定直接實作出助教提供的 Zero Cost algorithm 來降低 miss rate。

論文方面，演算的方法是看得懂的。但 zero cost algorithm 為什麼這樣挑 index 就可以降低 miss rate，我短時間內還沒研究出來。但根據論文的敘述與實驗數據顯示，zero cost algorithm 比傳統 LSB 的方式挑 index bit 來的佳。