108062608_黃柏翰_project_report

C. A. Final project:
The cache behavior simulation

目的：

這次的期末 project 目的是要找出在一連串的 memory access 最佳的 cache indexing bit 以達到最低的 miss。

演算法：

演算法分為三大部分:

A. 先找出不同的 index bit,　1. LSB　2.則是參考老師給的 paper(“Zero Cost Indexing for Improved Processor Cache Performance”)
B. 模擬一連串的 memory access 在 A 中找到的不同 index bit 下 cache 的 hit/miss 情形
C. 將 performance 較好的 index 寫入 index.rpt (助教要求的檔案)

程式碼：

A. 定義所需的函式庫，fstream 用來讀檔，cmath 以及 math.h 用來做一些數學計算，資料結構部分：project 中大多以 vector 來實作

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <math.h>
#include <vector>
```

B. 定義 struct，用來記錄（hit/miss,是否需要 replacement），（存最好結果）

```
// For checking hit/miss, replace or not
struct Hit_Replace{
    bool Hit;
    bool Replace;
};

// For file writing result
struct Result{
    vector <int> Index_bits;
    vector <string> Prediction;
    int Cache_miss;
};
```

C. 首先透過 argument 讀入引數並求出 offset bit, index bit, tag bit 數，為了因應

不同 index bit 所帶來的 Tag bit，所以加了 Address bit 來判斷

```
int main(int argc, char *argv[])
{
    string Cache_org, Reference_list, output_file;
    Cache_org = (string)argv[1];
    Reference_list = (string)argv[2];
    output_file = (string)argv[3];

    // for for loop, Total cache miss count
    int i, j, Total_cache_miss_count;

    // Read cache spec
    int Address_bits, Block_size, Cache_sets, Associativity;
    read_cache(Address_bits, Block_size, Cache_sets, Associativity, Cache_org);

    // Calculate info
    int Offset_bit_count, Indexing_bit_count, Tag_bit_count;
    Offset_bit_count = int(log2f(Block_size));
    Indexing_bit_count = int(log2f(Cache_sets));
    Tag_bit_count = Address_bits - Offset_bit_count - Indexing_bit_count;
    // for recording which bit have not been used (0 non, 1 used)
    // in access mode
    vector <int> Addressing_bits(Address_bits);
```

D. 讀 cache reference history 檔案，再來分別求 LSB，Zero_cost(參考 Zero Cost Indexing for Improved Processor Cache Performance 此 paper 的 index)

```
// Read index record
vector<string> index_history;
read_index_record(index_history, Reference_list);


// May be for LSB / Zero_cost indexing mode
// LSB for access mode
vector <int> LSB_index;
get_LSB_index(LSB_index, Indexing_bit_count, Tag_bit_count);
// Saving the redult of it
Result LSB_result;


// Zero_cost need extra information
vector<float>  Q(Address_bits - Offset_bit_count);
vector< vector<float> > Correlation(Address_bits - Offset_bit_count);
// if bit at set - >1 , otherwise - > 0
vector<int> AtSet(Address_bits - Offset_bit_count);
// result of sub optimal indexing, start from Zero_index.at(0)
// access mode
vector<int> Zero_index;
// Saving the redult of it
Result Zero_result;
prepare(index_history, Q, Correlation, AtSet);
Zero_cost(Q, Correlation, Zero_index, AtSet, Indexing_bit_count, Offset_bit_count);
```

E. 定義在 Cache reference 時所需的 Occupy(是否存有 Tag), Tag(那格裡面有什麼 Tag), NRU bit(該格的 NRU bit 為多少)，以及統計該 set 的 NRU counter，最後是定義由 cache reference history 求出的在查找時的 Set 以及 Tag

```
// Create needed vector, size == whole cache size
// For the cache block been occupied or not (0 for non, 1 for occupy)
vector< vector<int> >  Occupy(Cache_sets);
// For Tag in each cache block
vector< vector<string> >  Tag(Cache_sets);
// For NRU bits either 1 or 0
vector< vector<int> >  NRU(Cache_sets);
// NRU_counter for number of NRU set
vector<int>  NRU_counter(Cache_sets);
Setup_Essential(Occupy, Tag, NRU, NRU_counter, Cache_sets, Associativity);

// Checking Set
int Set;
// for checking with the tag in cache
string reference_Tag;
// Recording if the data hit/miss and replace or not
Hit_Replace hr;

// Get index_bit, Tag_bit in access mode (different indexing method, LSB, zero_cost here)
vector<int> Indexing_bits(Indexing_bit_count), Tag_bits(Tag_bit_count);
```

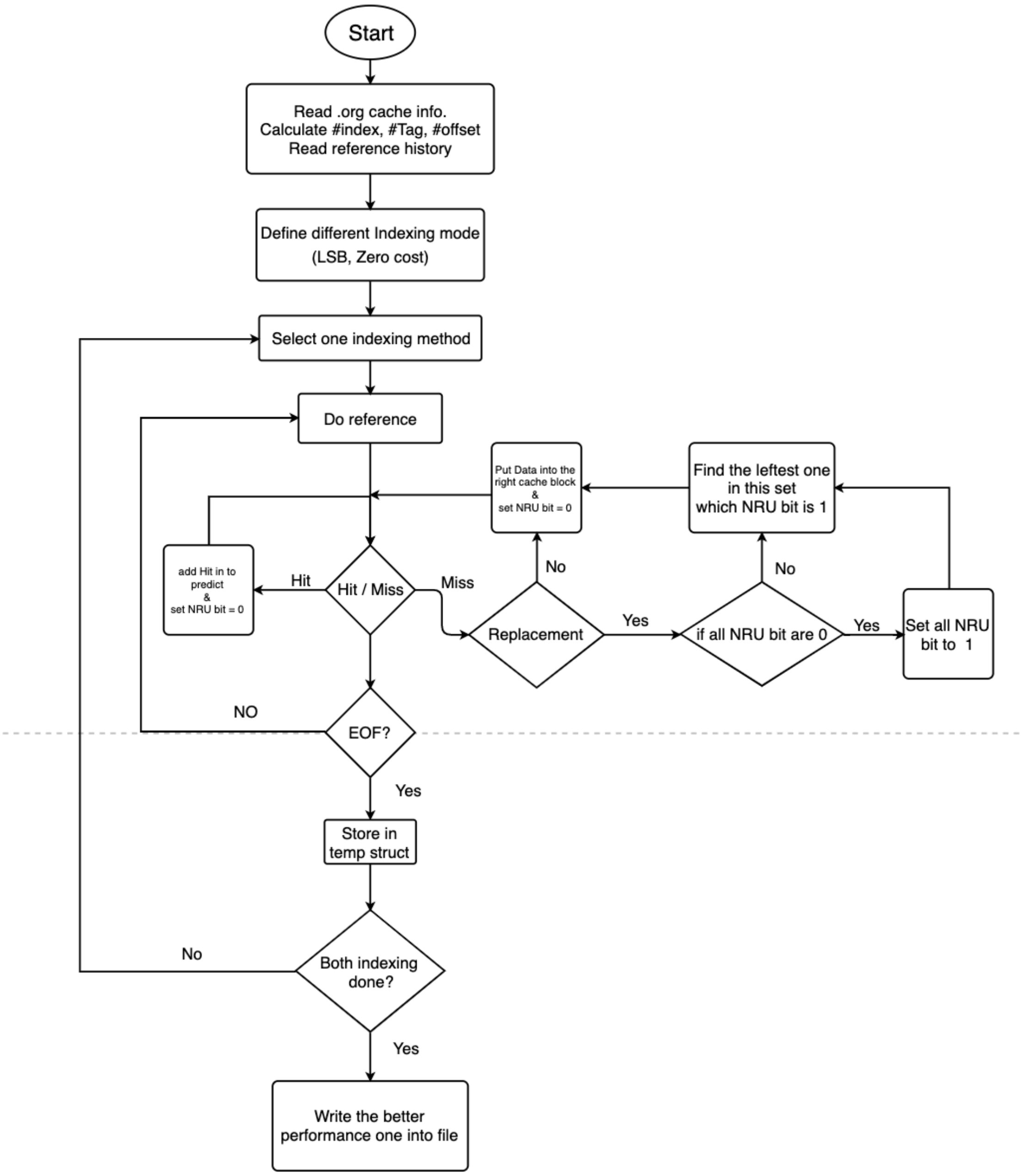
F. 程式的最後會根據所使用的不同 indexing 執行 Set 查找，比對 Tag 以及 Cache Replacement，最後根據不同的 indexing 寫入 A 步驟中定義好的 struct 中，最後比較出好的結果，寫入檔案

```cpp
int mode, isZero;
for (mode = 0; mode < 2; mode++){
    // if indexing method is zero bit;
    // 0 -> no, 1 -> yes
    isZero = mode;
    Total_cache_miss_count = 0;
    // Reset the Addressing bits
    Reset_Addressing_bits(Addressing_bits, Offset_bit_count);
    get_index_Tag(Indexing_bits, Tag_bits, Offset_bit_count, Indexing_bit_count, Tag_bit_count, Address_bits, isZero, Addressing_bits, Zero_index, LSB_index);

    Reset_Essential(Occupy, Tag, NRU, NRU_counter, Cache_sets, Associativity);


    // Do reference and Cache Replacement
    for (i = 2 ; i < index_history.size()-1 ; i++)
    {

        // Decide which set in cache to look up
        Set = find_set(Indexing_bits, Indexing_bit_count, index_history.at(i));


        // get reference_Tag and check_ache return hit/other
        get_reference_Tag(reference_Tag, Tag_bits, index_history.at(i));

        // check if Hit, so we must pass a Tag to check
        // Update the content of Occupy.at(Set), Tag.at(Set), NRU.at(Set), reference.at(Set), NRU_counter, reference_counter
        hr = Check_cache(Occupy.at(Set), Tag.at(Set), NRU.at(Set), NRU_counter.at(Set), reference_Tag);

        // Append Hit / Miss to predict_result
        update_predict_result(Zero_result.Prediction, LSB_result.Prediction, hr, Total_cache_miss_count, mode);

        // Since we have to replace to go on, do cache replacement algorithm
        if (hr.Replace == true){
            Cache_Replacement(Tag.at(Set), NRU.at(Set), NRU_counter.at(Set), reference_Tag, Associativity);
        }
    }
```

```cpp
        // LSB
        if (mode == 0){
            LSB_result.Index_bits = Indexing_bits;
            LSB_result.Cache_miss = Total_cache_miss_count;
        }
        else
        {
            Zero_result.Index_bits = Indexing_bits;
            Zero_result.Cache_miss = Total_cache_miss_count;
        }
    }
    if (Zero_result.Cache_miss < LSB_result.Cache_miss){
        // write file with a better result
        File_writing(Address_bits, Block_size, Cache_sets, Associativity, Offset_bit_count, Indexing_bit_count,
                    index_history, Zero_result, output_file);
    }
    else{
        // write file with a better result
        File_writing(Address_bits, Block_size, Cache_sets, Associativity, Offset_bit_count, Indexing_bit_count,
                    index_history, LSB_result, output_file);
    }

    return 0;
}
```

Ps. 圖片只顯示部分的程式碼，project 中的程式碼盡量以呼叫 function 來減少主程式的程式碼量，因為 sub function 數量較多，怕影響整體報告閱讀，所以沒附上

程式流程圖：


Start
Read .org cache info.
Calculate #index, #Tag, #offset
Read reference history
Define different Indexing mode
(LSB, Zero cost)
Select one indexing method
Do reference
Put Data into the right cache block & set NRU bit = 0
Find the leftest one in this set which NRU bit is 1
add Hit in to predict & set NRU bit = 0
Hit
Hit / Miss
Miss
No
No
Replacement
Yes
if all NRU bit are 0
Yes
Set all NRU bit to 1
NO
EOF?
Yes
Store in temp struct
No
Both indexing done?
Yes
Write the better performance one into file