0810740 張又仁

Computer Organization

Experiment background:

這一次的實驗主要是在模擬程式執行的時候，除了 CPU 運算的速度外，另外一個很重要的就是 memory cache 的存取，而其中 memory 的存取在 miss 的時候因為 penalty 很大，因此我們會盡量希望 Miss rate 下降，這樣的話程式執行的速度才能盡量達到穩定快速。
而 Miss 基本上可以分成三個種類，根據課本:

- **Compulsory misses**: These are cache misses caused by the first access to a block that has never been in the cache. These are also called **cold-start misses**.
- **Capacity misses**: These are cache misses caused when the cache cannot contain all the blocks needed during execution of a program. Capacity misses occur when blocks are replaced and then later retrieved.
- **Conflict misses**: These are cache misses that occur in set-associative or direct-mapped caches when multiple blocks compete for the same set. Conflict misses are those misses in a direct-mapped or set-associative cache that are eliminated in a fully associative cache of the same size. These cache misses are also called **collision misses**.

第一種是 compulsory miss，主要是發生在初始 cache，沒有 block 的 tag 符合而且剩下一些 block 的 valid bit 都還沒啟動的狀況下。第二種是 capacity miss，是在所有 block 的 valid bit 都啟動的狀況下，卻因為執行一個程式所需要的 block 數量大於 cache 的空間，因此必定會發生需要 replacement 的 Miss。第三種是 Conflict miss，主要是把焦點放在單獨的 set 裡面，當 set 滿了的時候，必須 replace block 在特定的 set 所發生的 miss。
我自己對於第二種跟第三種的理解是，發生 conflict miss 不代表發生 capacity miss，因為有可能所有指令都只發生在同個 set，這樣就不是 capacity 的問題，而 capacity miss 發生的話，倘若不是 fully associative cache，那麼也可以歸類在 conflict miss，而我覺得比較重要的應該是，如果我們知道一個 memory 執行程式的時候只發生 conflict miss 而沒發生 capacity miss，那就表示執行這個程式如果需要 Miss rate 降低，所用的 cache 最好是高 associativity 的，這樣 conflict 就會減少，而如果一個程式只發生 capacity miss，conflict miss 沒什麼發生，那麼最好 cache 的 block size 就大一點，這樣就可以利用 Spatial locality 的特性降低 Miss rate.

不過在這次的實驗僅僅考慮 Miss rate，因此提高 associativity 造成的 Hit penalty 上升進而下降運行速度的部分並不討論。

Experiment theories:

因為這次探討的對象是 memory，因此我們必須知道有什麼樣的面相與 Miss rate 有關，可以讓我們的 Memory 運行速度提高。根據課本:

- **Temporal locality** (locality in time): if an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.
- **Spatial locality** (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon. For example, when you brought out the book on early English computers to find out about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you also brought back that book and, later on, found something useful in that book. Libraries put books on the same topic together on the same shelves to increase spatial locality. We'll see how memory hierarchies use spatial locality a little later in this chapter.

主要有 Temporal locality 與 Spatial locality，前者會與 cache 儲存近期使用到的地址有關，後者與 cache 儲存所用的 block 大小有關。
這次實驗主要模擬 cache 在 miss rate 的表現，並沒有討論實際運行的速度，並且探討 block size 與 associativity 對於 Miss rate 的影響，同時 replacement 是使用 LRU 的方式來達成，輸入只會有 cache size 跟 block size 還有 associativity，因此我們要探討 Miss rate 的話，必須先知道我們所擁有的 set 數量，進而決定 tag bit 跟 index bit，以利我們判斷 conflict miss 的發生與否，根據:

Since the block size is 2^m words (2^{m+5} bits), and we need 1 bit for the valid field, the number of bits in such a cache is

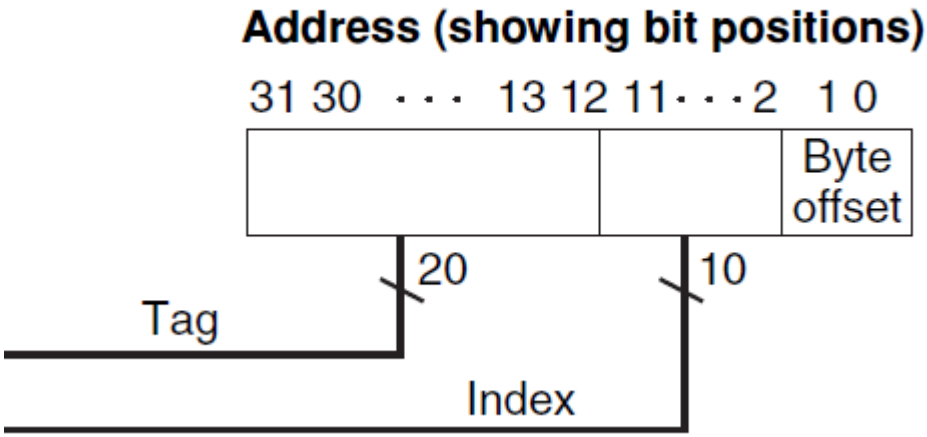
$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m).$$

這是 direct mapping 的公式，但是只要把 direct mapping 想成 associativity=1，我們就可以改寫出我們需要的公式，得到一個 general 的概念與解法(因此以下統一用 set 的形式寫)，根據課本:

(Block number) modulo (Number of *sets* in the cache)

Since the block may be placed in any element of the set, *all the tags of all the elements of the set* must be searched. In a fully associative cache, the block can go anywhere, and *all tags of all the blocks in the cache* must be searched.

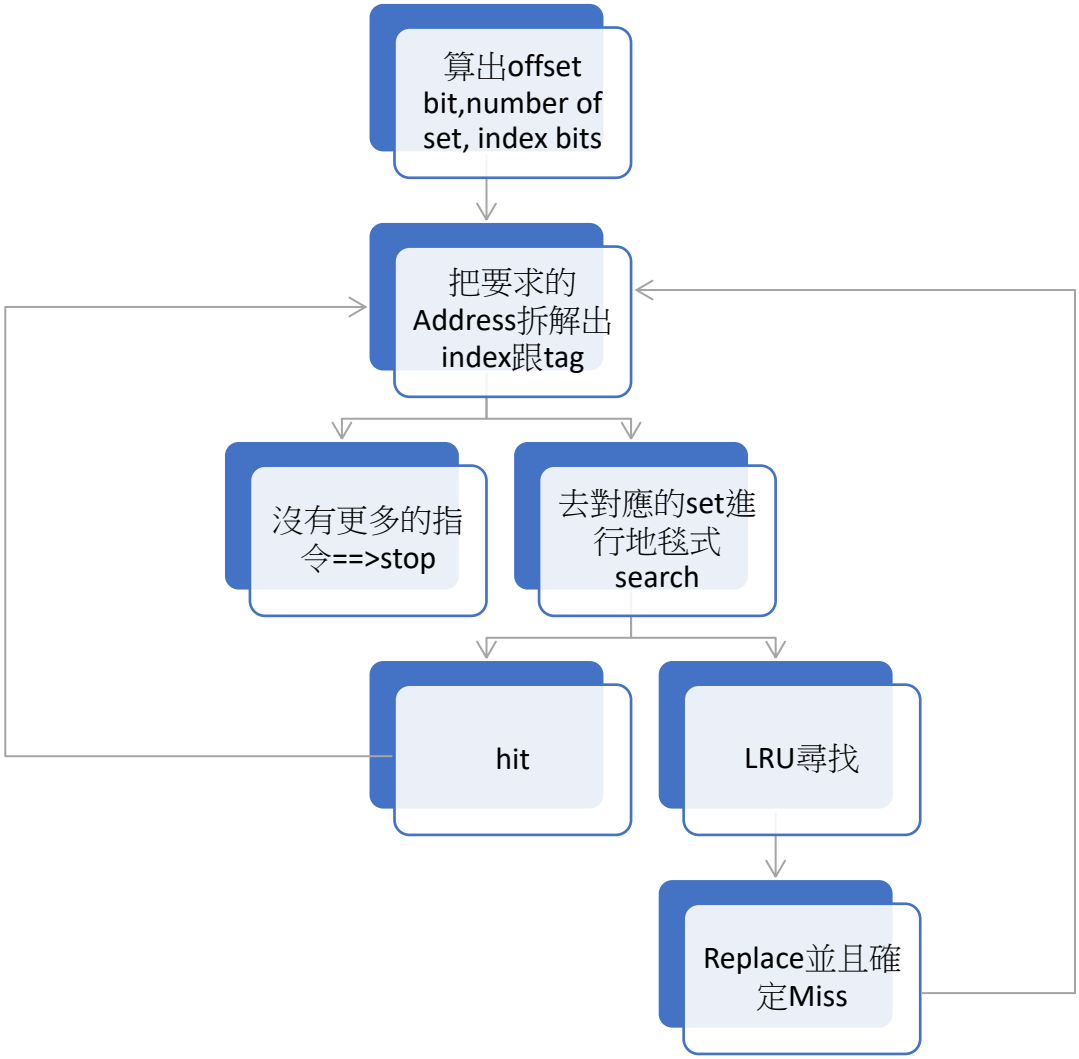
我們可以知道 set 的概念就是讓 hit time 久一點，search 一個特定 set 裡面所有的 block，但是至少 Miss rate 會因為我們對於同個 index 的 block 可以存放的個數上升，就可以利用 Temporal locality 的特性來減低 Miss rate。
因此我們需要的公式如下:
Offset bit:log2(block_size)
Number of block:cache_size/block_size(or cache_size>>Offset bit)
Number of set:Number of block/associativity
Index bit:log2(Number of set)
而根據課本:



我們可以知道 cache 運作時，address 由左到右會被區分成 tag, index, offset 這三個大區塊。
故我們由上面的探討可以算出需要的 set(單次 search 的對象)數量以後，接著就是要根據要求的 Address，來看有沒有在對應的 set 裡面找到要求的 address，而要知道，Block 本身就是利用 Spatial locality 的特性把數個地址抓進來，因此當我們想要知道有沒有在 cache 裡面時，找的是”block”，故我們探討”Miss rate”的時候，我們不需要管找到 Block 以後根據 offset 去做的處理，所以要求的 address>>offset_bits=address1，接著我們如果要探討有沒有 hit，就要利用 index 跑去對應的 set 裡面尋找 tag，所以 address1&(Number of set-1)=address2，-1 的原因是我們在使用 bit 表示數字的時候，是從 0 開始計算的，與我們計算個數從 1 開始是不一樣的，因此這是一個要注意的點。

得到對應的 index(address2)以後，我們就可以把 address1>>index_bits，找到我們要找的 address 的 tag，然後拿去與對應 index 的 set 去做一個比較，找到的話就 hit，沒有找到的話就要根據 LRU 找出要被取代的 block(因為僅僅考慮”Miss rate”沒有考慮 Miss 來源，因此無論 valid bit or not 都是當作取代來處理，這樣寫程式比較 general)。

因此，總結我們可以把這次實驗的模擬 coding 看成以下的 flowchart:



Experiment content:

這次的實驗一共有 5 個檔案，一個主要進行模擬的 cpp 檔(direct_map_cache.cpp)，一個在 linux 下可以快速生成編譯檔的 makefile，一個我用來當作我做實驗用的地址檔(Trace.txt)，以及兩個我把生成的結果拿去 matlab 生成圖表的檔案(result1.fig,result2.fig)。

1. direct_map_cache.cpp

這個模擬檔會把我上面所寫的流程圖實現出來。

Coding:

```
#include <getopt.h>
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <string>
#include <vector>
using namespace std;

struct cache_content { // consider lru
  bool v;
  unsigned int tag;
  unsigned int lasttime;
  //    unsigned int    data[16];
};

const int K = 1024;

void simulate(int cache_size, int block_size, int asso, string
&test_file_name,
              int test) {
  FILE *fp = fopen(test_file_name.c_str(), "r"); // read file

  if (!fp) {
    cout << "Test file doesn't exist\n";
    return;
  }

  unsigned int tag, index, x;
  // 2^n*(2^m*(one block size)+(one block size-n-m)+1) is total size
in one
  // cache
  // assume asso=2^x,x comes from nature number
  int offset_bit = (int)log2(block_size);
  int line = cache_size >> (offset_bit) / asso; // num of set
  int index_bit = (int)log2(line);//index of set
```

```
  cache_content **cache = new cache_content *[line];
  vector<int> instruc_hit;
  vector<int> instruc_miss;
  for (int i = 0; i < line; i++)
    cache[i] = new cache_content[asso];
  cout << "cache line:" << line << endl;
  for (int i = 0; i < line; i++)
    for (int j = 0; j < asso; j++) {
      cache[i][j].v = false; // valid bit //calculated for compulsory
miss
      cache[i][j].lasttime = 0;
      cache[i][j].tag = 0;
    }
  unsigned int total = 0, miss = 0;
  while (fscanf(fp, "%x", &x) != EOF) {
    total++;
    cout << hex << x << " ";
    index =
        (x >> offset_bit) & (line - 1);  // line-1==>because from 0
to 2^line-1
    tag = x >> (index_bit + offset_bit); // index get set num
    unsigned int target = 0;
    for (int i = 0; i < asso; i++) { // set check
      if (cache[index][i].v && cache[index][i].tag == tag) {
        cache[index][i].v = true; // hit
        target = i;               // find out
        instruc_hit.push_back(total);
        break;
      } else if (cache[index][i].lasttime <
cache[index][target].lasttime) {
        // process replacement
        target = i;
      }
    }
    if (cache[index][target].v == false) {
      // compulsory miss
      miss++;
      instruc_miss.push_back(total);
```

```
    } else if (cache[index][target].tag != tag) {
      // conflict/capacity miss
      miss++;
      instruc_miss.push_back(total);
    }
    cache[index][target].lasttime = total;
    cache[index][target].tag = tag;
    cache[index][target].v = true;
  }
  fclose(fp);

  fp = fopen("Result.txt", "a"); // for people read
  // basic information
  fprintf(fp, "Cache size:%d\nBlock size:%d \nAssociativity:%d\n",
cache_size,
          block_size, asso);
  // asked information
  fprintf(fp, "Miss rate:%f%%\n", 100.0 * miss / total);
  printf("\nMiss rate:%f%%\n", 100.0 * miss / total);
  fprintf(fp, "Hits instructions:");
  printf("Hits instructions:");
  for (int i = 0; i < instruc_hit.size(); i++) {
    fprintf(fp, "%d", instruc_hit.at(i));
    i != (instruc_hit.size() - 1) ? fprintf(fp, ",") : NULL;
    printf( "%d", instruc_hit.at(i));
    i != (instruc_hit.size() - 1) ? printf(",") : NULL;
  }
  fprintf(fp, "\nMisses instructions:");
  printf("\nMisses instructions:");
  for (int i = 0; i < instruc_miss.size(); i++) {
    fprintf(fp, "%d", instruc_miss.at(i));
    i != (instruc_miss.size() - 1) ? fprintf(fp, ",") : NULL;
    printf("%d", instruc_miss.at(i));
    i != (instruc_miss.size() - 1) ? printf(",") : NULL;
  }
  fprintf(fp, "\n-----------------------------------------\n");
  printf("\n-----------------------------------------\n");
  fclose(fp);
```

```
  if (test == 1) {
    fp = fopen("Result_data1.csv", "a"); // for plotting data
    fprintf(fp, "%d,", cache_size/K);
    fprintf(fp, "%d,", block_size);
    fprintf(fp, "%d,", asso);
    fprintf(fp, "%f\n", 100.0 * miss / total);
    fclose(fp);
  }
  else if (test == 2) {
    fp = fopen("Result_data2.csv", "a"); // for plotting data
    fprintf(fp, "%d,", cache_size/K);
    fprintf(fp, "%d,", block_size);
    fprintf(fp, "%d,", asso);
    fprintf(fp, "%f\n", 100.0 * miss / total);
    fclose(fp);
  }
  for (int i = 0; i < line; i++)
    delete[] cache[i];
  delete[] cache;
  instruc_hit.clear();
  instruc_miss.clear();
}

int main(int argc, char **argv) {
  string test_file_name;
  int cache_size = 4;
  int block_size = 16;
  int associativity = 1;
  int current_option;
  while ((current_option = getopt(argc, argv, "f:c:b:a:")) != EOF) {
    switch (current_option) {
    case 'f': {
      test_file_name = string(optarg);
      break;
    }
    case 'c': {
      cache_size = atoi(optarg);
      break;
```

```
    }
    case 'b': {
      block_size = atoi(optarg);
      break;
    }
    case 'a': {
      associativity = atoi(optarg);
      break;
    }
    }
  }

  // default simulate 4KB direct map cache with 16B blocks
  simulate(cache_size * K, block_size, associativity,
test_file_name,0);
  // test1
  //for (int i = 1; i <= 512; i *= 2)
  //  for (int j = 1; j <= 512; j *= 2)
  //    simulate(i * K, j, 1, test_file_name,1);
  // test2
  //for (int i = 1; i <= 512; i *= 2)
  //  for (int j = 1; j <=32; j *= 2)
  //    simulate(i * K, 32, j, test_file_name,2);
}
```

Features:

我修改了一點 direct_mapping 的 code，讓他變成符合我上面所推倒的 set 的 code，其中 cache 裡面會有一個 lasttime 紀錄上次要抓取這個地址是在程式的第幾行(假設這個程式只會由第一行跑到最後一行)，並且我們使用的 Address 是 Byte 為單位(與電腦在處理記憶體空間的單位一致)，可以看到我的 coding 正如上面的流程圖一樣分成三個大區塊，第一個區塊是先把基本資料，bit 樹還有 number of set 都先找出來並且生成以”block”為最小單位的陣列以及因為題目要求要找出 hit 跟 miss 的 instruction，因此特別創立的 vector。
第二區塊則是開始吃 address，然後拆解 Address 先找出 index，再根據 index 使用 for loop 來 Search 我們的 set，如果 hit 就把 target 設成對應的 i 然後 break，沒有 hit 就繼續根據 LRU 的特性來找到要被 replacement 的 target，接著判斷是屬於哪種 Miss，最後則是不管有沒有 miss 都把 target 改覆蓋一次(雖然會有點浪費執行時間，但主要是想 general coding)。

最後區塊則是把對應的 data 輸出成 txt 檔和 csv 檔，其中 Result.txt 檔會把相關的資本資訊及對應輸出的 Miss rate 寫成比較好易讀的樣子。而 Result_data1.csv 跟 Result_data2.csv 則是為了對應 spec 要求生成的兩個圖所創立的。

如果把最下面的注釋拿掉:

```
// default simulate 4KB direct map cache with 16B blocks
simulate(cache_size * K, block_size, associativity, test_file_name,0);
// test1
//for (int i = 1; i <= 512; i *= 2)
//  for (int j = 1; j <= 512; j *= 2)
//    simulate(i * K, j, 1, test_file_name,1);
// test2
//for (int i = 1; i <= 512; i *= 2)
//  for (int j = 1; j <=32; j *= 2)
//    simulate(i * K, 32, j, test_file_name,2);
```

因為我在 simulate 函數的最後有放一個 test 的 int 值來判斷要不要生成 csv 檔，因此當我們把最下面兩個拿掉的時候，他會在把要求的 simulate 條件做完後，固定生成根據這個 Trace 檔我們從 cache 1KB~512KB 及 block size 從 1~512byte 的第一個 csv(associativity 固定為 1)，以及 cache size 1KB~512KB 及 associativity 從 1 到 32 的第二個 csv(block size 固定為 32bytes)。(生成的 csv 檔在 cache size 單位是 KB)。

以下為範例 output:

```
bop@unix[18:44:39][~]> ./cache_simulator -f Trace1.txt -c 1 -b 32 -a 2
cache line:256
bfa437cc bfa437c8 bfa437c4 bfa437c0 bfa437bc bfa437b8 bfa437b8 bfa43794 b8088ea8
 b8088eac
Miss rate:40.000000%
Hits instructions:2,3,4,6,7,10
Misses instructions:1,5,8,9
```

2. makefile

在 Linux 系統下，只要輸入 make command 就可以把上面的 cpp 檔編譯。

Coding:

```
all:direct_map_cache.cpp
    g++ direct_map_cache.cpp -o cache_simulator
clean:
    rm cache_simulator
    rm Result.txt
    rm Result_data1.csv
    rm Result_data2.csv
```

Features:

只需要輸入一次 make 就可以直接生成出編譯檔，並對編譯檔進行 command 指

令，如附圖:

```
oop@unix[17:21:21][~]> make
g++ direct_map_cache.cpp -o cache_simulator
direct_map_cache.cpp: In function ‘ void simulate(int, int, int, std::__cxx11::s
tring&, int)’ :
direct_map_cache.cpp:90:56: warning: converting to non-pointer type ‘ long int’
 from NULL [-Wconversion-null]
     i != (instruc_hit.size() - 1) ? fprintf(fp, ",") : NULL;
                                                        ^~~~
direct_map_cache.cpp:95:57: warning: converting to non-pointer type ‘ long int’
 from NULL [-Wconversion-null]
     i != (instruc_miss.size() - 1) ? fprintf(fp, ",") : NULL;
                                                         ^~~~
oop@unix[17:21:40][~]> ./cache_simulator -f Trace1.txt -c 1 -b 32 -a 2
cache line:256
bfa437cc bfa437c8 bfa437c4 bfa437c0 bfa437bc bfa437b8 bfa437b8 bfa43794 b8088ea8
 b8088eac oop@unix[17:21:42][~]> ls
cache_simulator       makefile    Trace1.txt
```

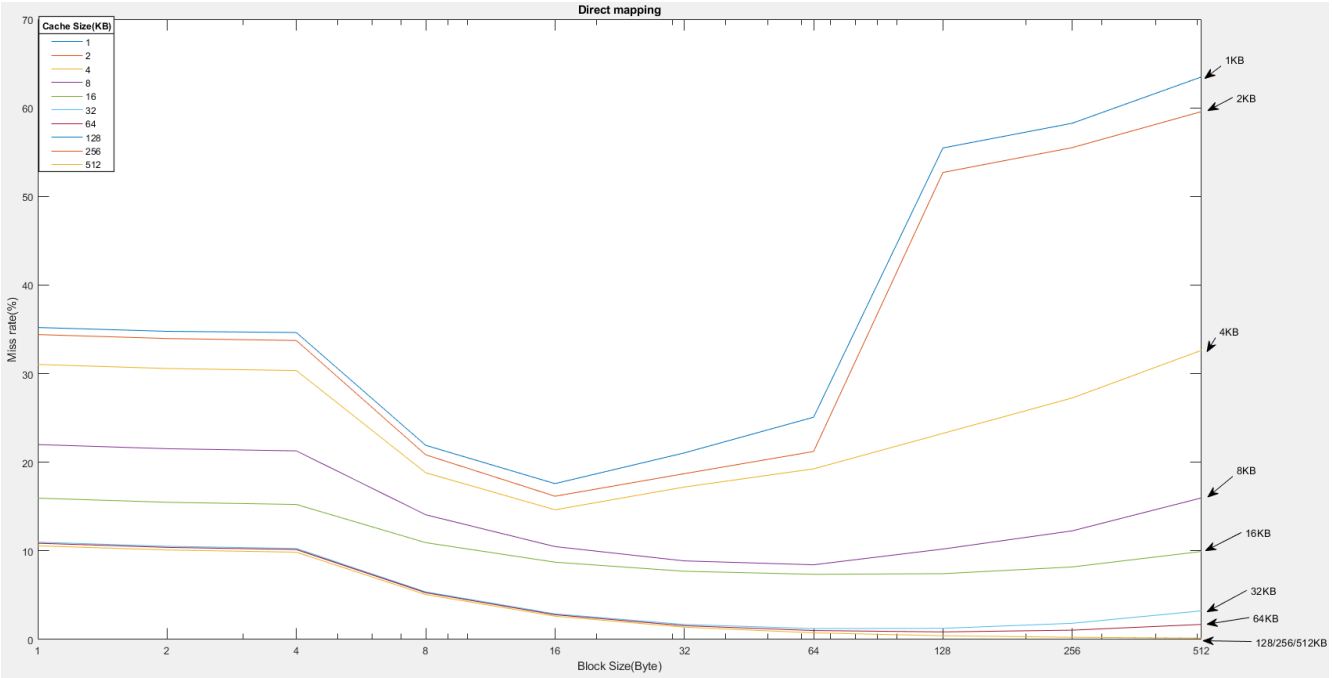
並且生成完後可以使用指令 make clean 來清除剛剛生成出的檔案，如附圖:

```
oop@unix[17:21:43][~]> make clean
rm cache_simulator
rm Result.txt
rm Result_data1.csv
rm: cannot remove 'Result_data1.csv': No such file or directory
makefile:4: recipe for target 'clean' failed
make: *** [clean] Error 1
oop@unix[17:21:48][~]> ls
direct_map_cache.cpp  makefile  Trace1.txt  Trace.txt
oop@unix[17:21:49][~]>
```

3. trace 檔，Trace.txt 只要是我在下面結果呈現所使用的 Trace 檔，因為我在使用 Trace1.txt 發現指令太少，以至於 locality 的特性無法看出區別，因此我就利用 Trace.txt 龐大的指令數量來進行本次的模擬。
4. result1.fig 這個檔案點開後可以直接把我在下面的第一個結果顯示出來。
5. result2.fig 這個檔案點開後可以直接把我在下面的第二個結果顯示出來。

Experiment result:

1. 固定為 direct_mapping，但是觀察 cache_size 與 block_size 變化對 Miss rate 造成的影響:



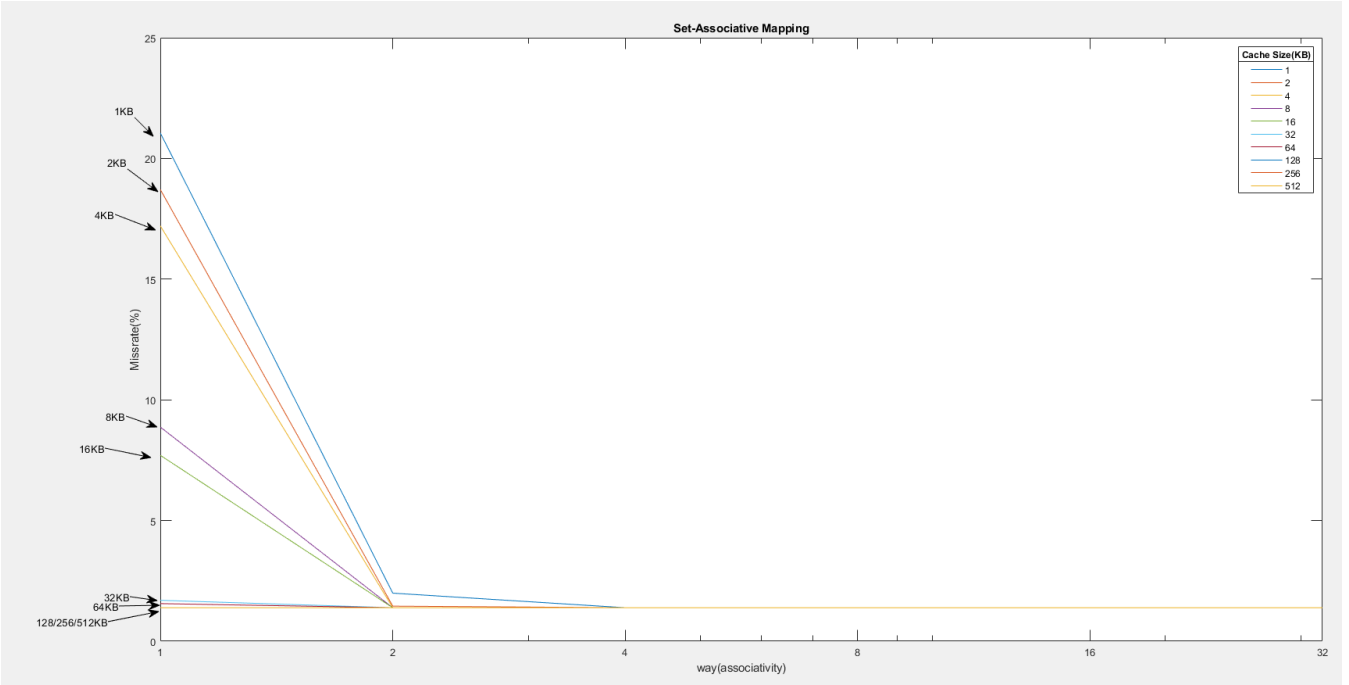
Result1

我們可以看到，在 cache size 8KB 以前，Miss rate 會先在 block size 大約為 16Byte 達到最低點，接著當 Block size 上升，miss rate 也跟著上升，為什麼會這樣呢?

我們可以做個簡單的運算與比較，當 Cache size 是 4KB 時，block size 分別測試 2Byte 與 16 byte 還有 256 Byte，我們可以算出 number of sets(這裡 set 裡面僅有一個 block)分別是 2^11 以及 2^8 以及 2^4，可以大概發現，在 direct_mapping 下，對於 cache size 比較小的狀況，如果 Block size 太小，那麼就會因為無法好好利用 Block Spatial locality 的特性，而導致 Miss rate 比較高，但是假若提升一些 Block Spatial locality，就可以調降 Miss rate，不過為什麼在 Block size 為 256 Bytes 時，反而卻大大的抬高 Miss rate 呢?可以發現在 Block size 為 256Bytes 時，我們僅有 16 個 Block，而我們卻無法保證 Trace.txt 裡面的程式會用到的 block 區域只有 16 個，因為 Block 的優勢僅僅是把鄰近特定 address 的一個大區塊存下來，但是程式會用到的 address 往往卻不一定全部都有”相鄰性”，因此假如我們把會用到的 Address 的周圍會被 Block 存儲的區塊作紀錄，就可以發現當會用到的 address 繁多且又沒有重複性時，很容易就超過 16block 所形成的區域。

而在 cache size 比較高的情況，block size 的增加導致 number of set 數量降低的影響就會下降(因為在所有條件固定僅改變 cache_size 的狀況下，number of set 會隨著 cache size 增加而增加，進而可以下降 block 數量不夠的問題)。而其中可以看到倘若我們使用的 cache size 到達 512KB，block size 的增加降低 Miss rate 的範圍會變得更大，因此即便 Block size 到達了 512 Bytes，依舊維持非常低的 Miss rate.

2. 固定 Block size 為 32 Bytes，調控 cache size 與 associativity 來觀察 Miss rate 的變化:



可以看到，倘若我們固定 block size，在同樣的 cache size 下，提高 associativity，可以降低我們的 Miss rate，而且可以看到降低從 direct_mapping 到 set =2 的降低幅度最大，這與我們在課本上所看到的結果一致，這個下降主要是因為同個 index 下，可以存放的 block 數量增加了，這與上面 direct mapping+ block size=32 Byte 最大的差別是哪裡呢?答案就是我們把 block 分組的成員數量調升了，所以當我們進行 LRU 取代時，會是從兩個以上的 block 進行挑選，而不是直接把單一 block 取代掉，換句話說，associativity 的提高會讓我們使用 temporal locality 的特性被放大，所以長期來看，即便過了很久，仍可能重複執行的 address 還留在 cache 裡沒有被取代掉!但是當我們 associativity 調更高以後，就會因為這樣的特性已經被充分展現(畢竟僅僅需要重複的 Address 還留在 set 裡即可有這樣的特性)，而導致 Miss rate 無法下降很多，值得注意的是，與第一題不同，把 associativity 調很高，仍然不會有 Miss rate 上升的情形，這主要是因為 associativity 並不會改變 block 的數量，而我們 cache 處理記憶體的單位是 block，因此只要 block 數量不動，那麼可以容納的區域數量就不會受到影響➔應對 address 可能位在數個區域的程式能力不被影響。

最後可以發現當我們把 cache size 上升以後，Miss rate 也會下降，這主要與 block 數量的上升有關，因此 temporal locality 的特性就會被展現出來。

Problems you met and

solutions:

在這次的實驗最難的應該是要把 direct cache 的 coding 轉成 set associativity cache，因為這時候就很需要知道 set 與 index 之間是對應的關係，不過只要照著以下邏輯走:

必須知道 set number 才能知道 index bits，必須知道 block number 才能知道 set number，那我們就可以很輕易的把 set 的問題從 direct cache 開始想，一步一步推導出來了!

Summary:

根據上面的圖表，看起來好像 Cache size 越大或是 associativity 越大就越好，但那僅僅是針對 Miss rate 的部分，根據課本:

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

我們可以看到當我們考慮的是整體的執行速率時，cache size 跟 associativity 的增加都會導致 access time 的時間被拉長(例如我們在 set 要找目標 block 時，使用的 for loop 就是一個例子)，因此這次的模擬僅僅是針對 Miss rate，這還是必須要明白的一件是。

另外，因為這次作業要用 C 語言寫，因此在理解跟打 code 速度上都快比較多，感覺還蠻好的。

而在這課程打了 6 次的 Lab，我覺得每次都有一種讓我更明白課本上意思的感覺，因為我自己覺得我不是一個單單讀課本，考試就可以拿很高分的人，我讀課本時，會想到的問題跟推理真的很少，但是這些實作，反而會激勵我去用課本的東西推理跟理解更透徹，我覺得這真的是蠻好的學習過程，也讓我明白工程科系的科目都不能只是單單”讀”，更多的是理解及應用這些知識在實際的問題裡，反覆的思考與反覆的使用才能找到學習的盲點。