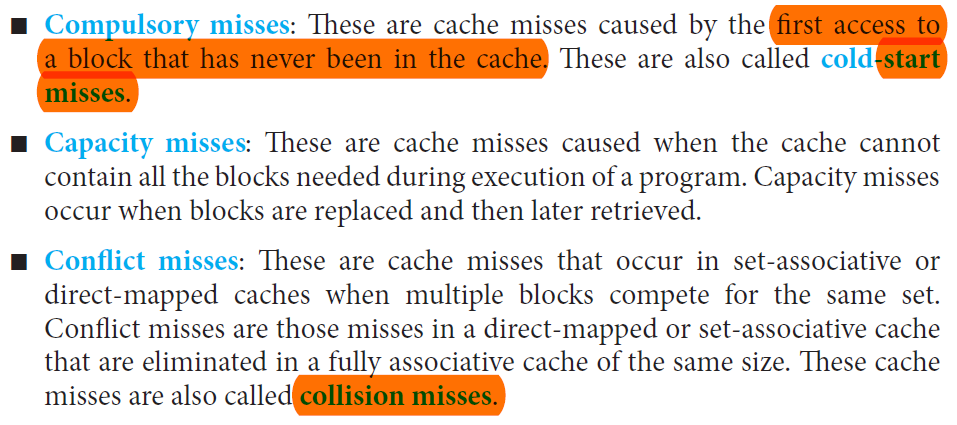
**0810740 張又仁**

**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:



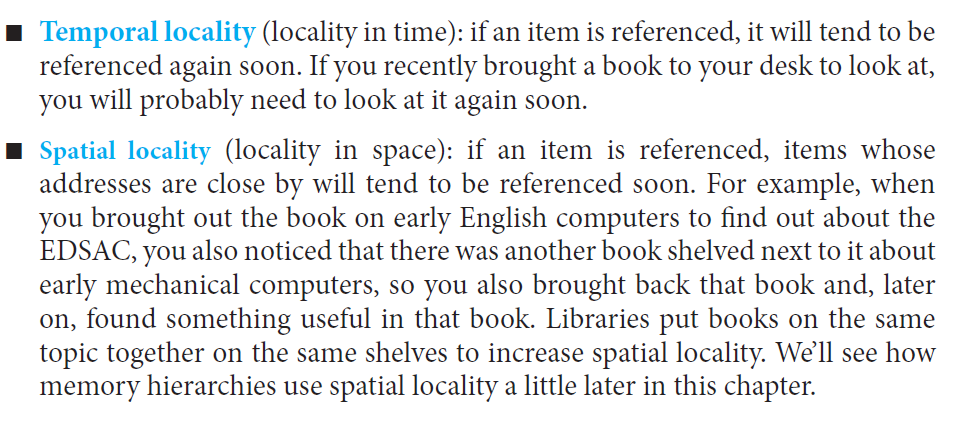
第一種是compulsory miss，主要是發生在初始cache，沒有block的tag符合而且剩下一些block的valid bit都還沒啟動的狀況下。第二種是capacity miss，是在所有block的valid bit都啟動的狀況下，卻因為執行一個程式所需要的block數量大於cache的空間，因此必定會發生需要replacement的Miss。第三種是Conflict miss，主要是把焦點放在單獨的set裡面，當set滿了的時候，必須replace block在特定的set所發生的miss。

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。

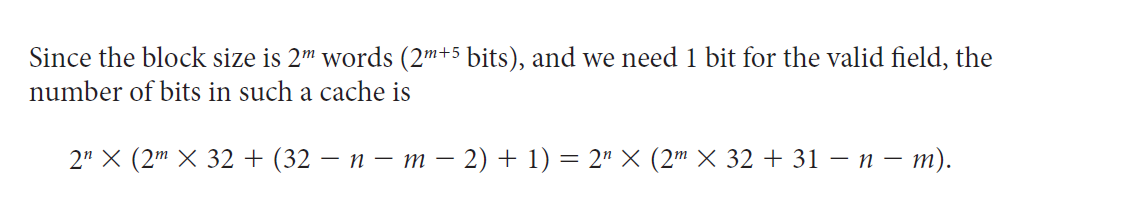
**Experiment theories**:

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:

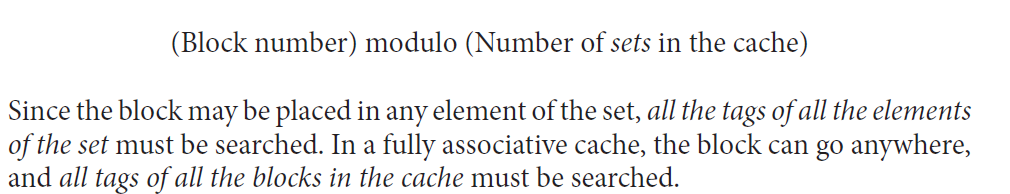


主要有Temporal locality與Spatial locality，前者會與cache儲存近期使用到的地址有關，後者與cache儲存所用的block大小有關。

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:



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:



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:

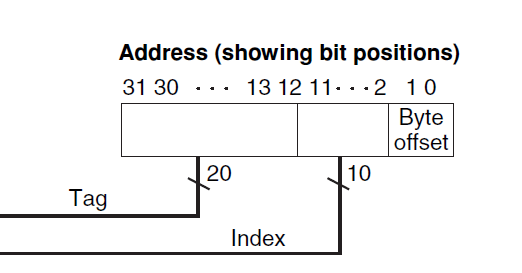
Offset bit:log2(block\_size)

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);

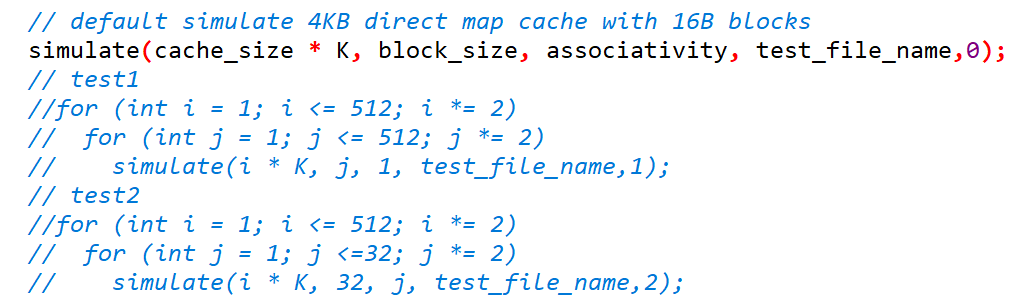
}

Features:

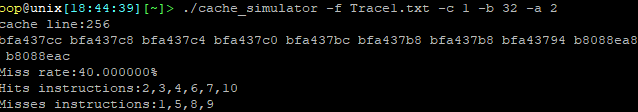
我修改了一點direct\_mapping的code，讓他變成符合我上面所推倒的set的code，其中cache裡面會有一個lasttime紀錄上次要抓取這個地址是在程式的第幾行(假設這個程式只會由第一行跑到最後一行)，並且我們使用的Address是Byte為單位(與電腦在處理記憶體空間的單位一致)，可以看到我的coding正如上面的流程圖一樣分成三個大區塊，第一個區塊是先把基本資料，bit樹還有number of set都先找出來並且生成以”block”為最小單位的陣列以及因為題目要求要找出hit跟miss的instruction，因此特別創立的vector。

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:

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:

1. 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

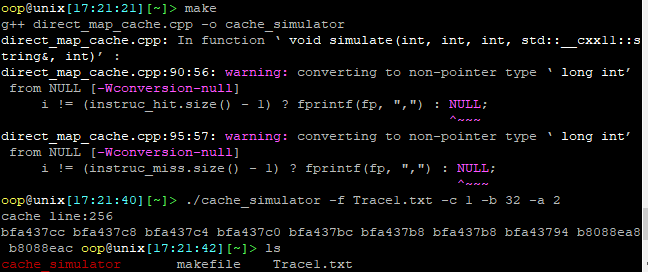
rm Result.txt

rm Result\_data1.csv

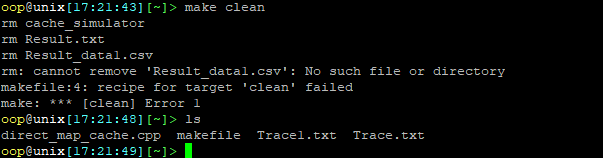
rm Result\_data2.csv

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:



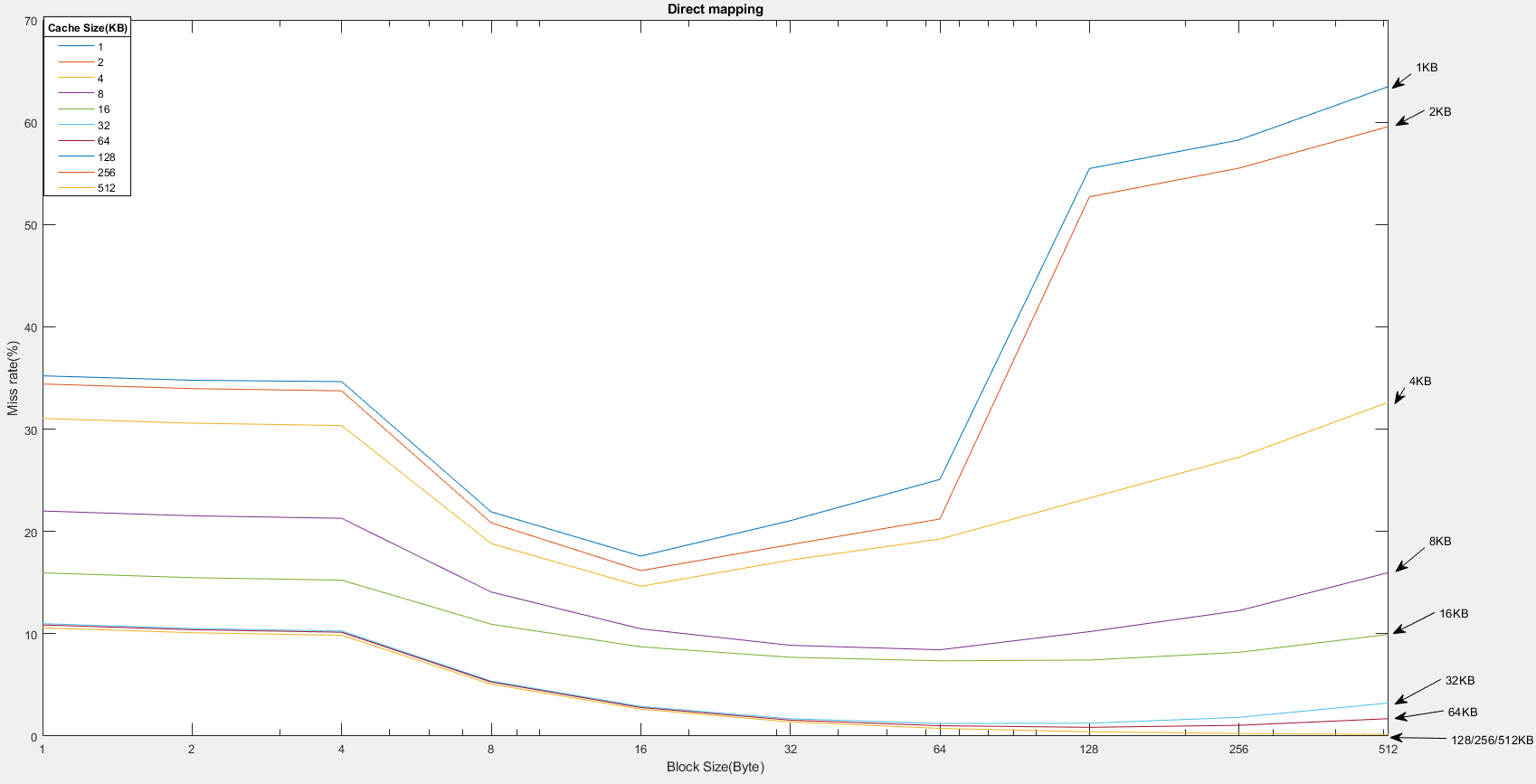
並且生成完後可以使用指令 make clean來清除剛剛生成出的檔案，如附圖:



1. 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。
2. 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。
3. 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:



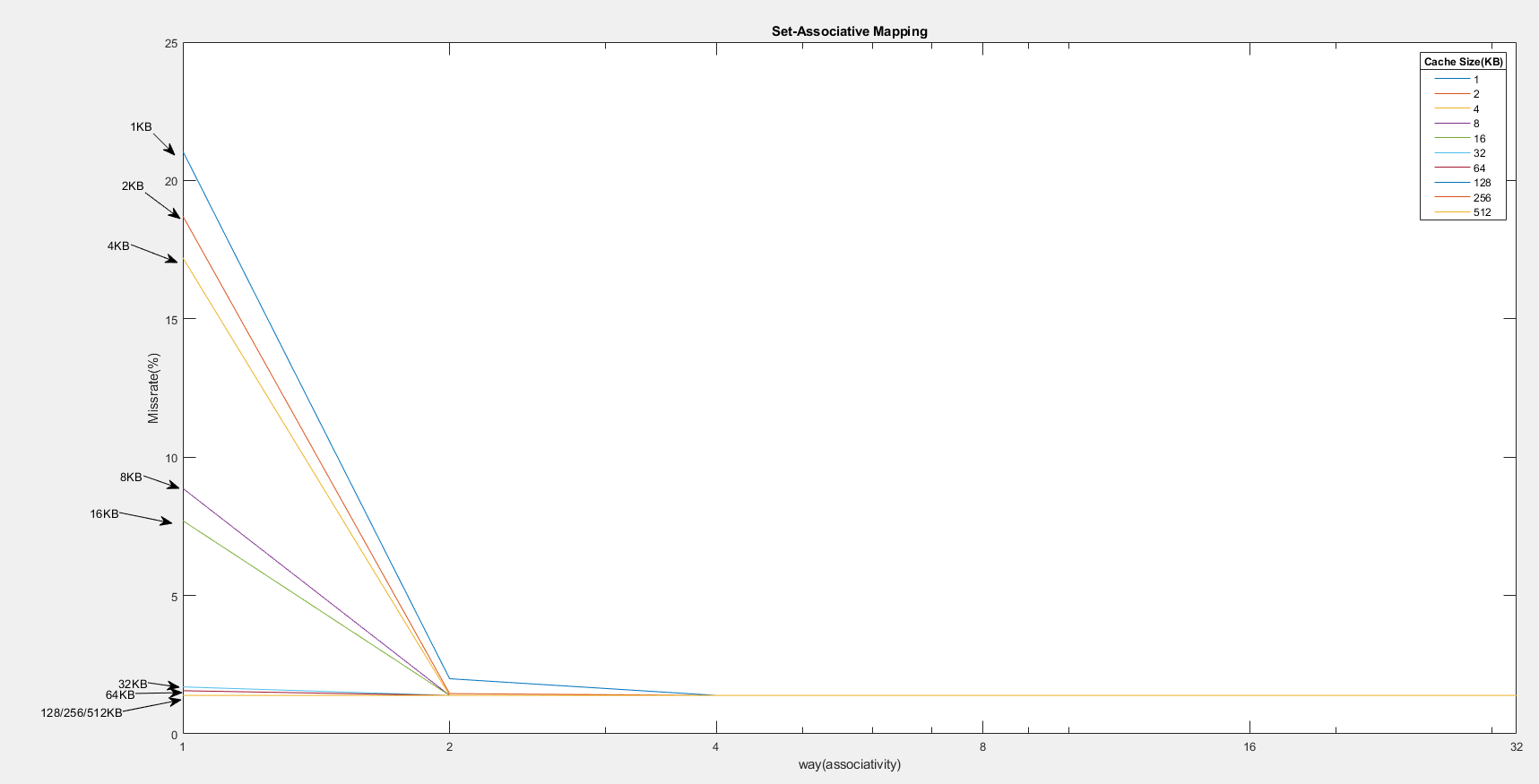
Result1

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.

1. 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。

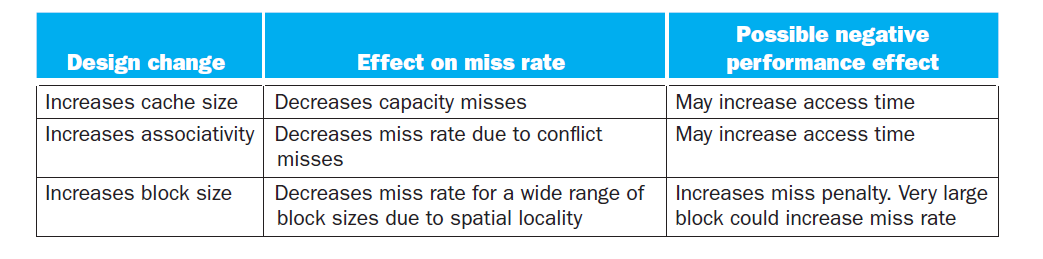
**Problems you met and solutions:**

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:



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。