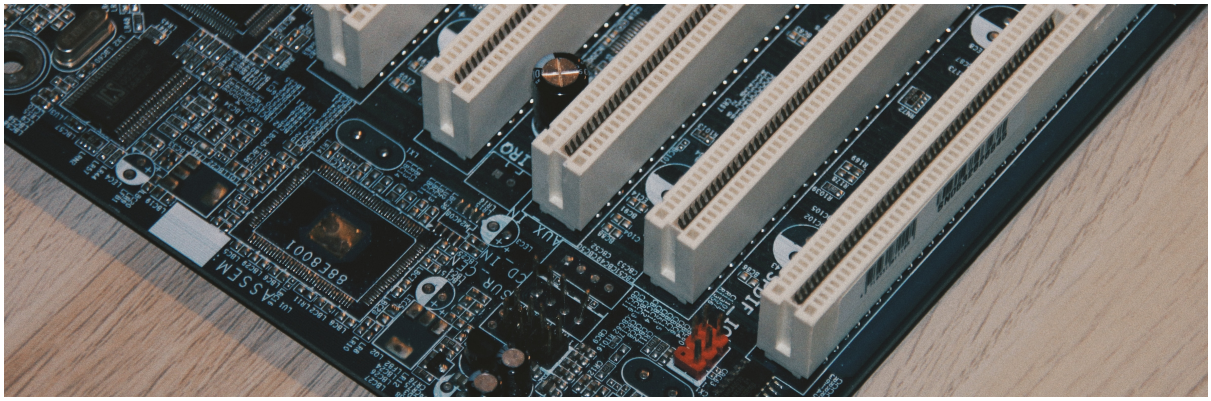
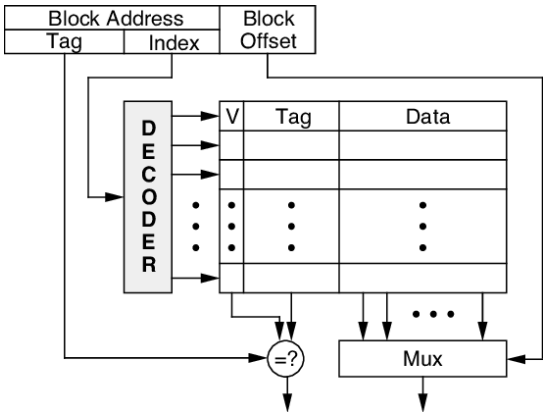


Lab 6: Cache Simulator

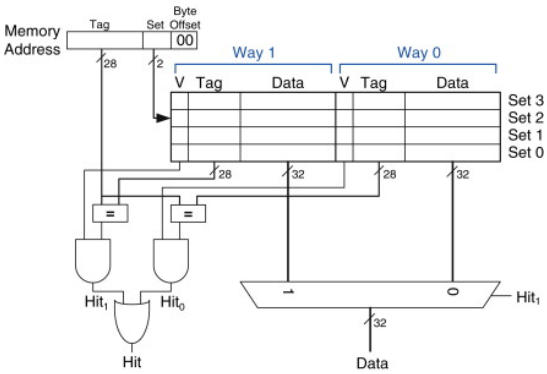
姓名：葉致廷/林廷澔
學號：0846013/0846011

Architecture diagram

- **direct_mapped_cache**



- **set_associative_cache(2-way example)**



Detailed description of the implementation

- **direct_mapped_cache**

  ▼ **Description**

  不管是ICACHE或是DCACHE，兩者在block size變大後，miss rate都會下降，這是因為spatial locality改善的緣故。另外cache size的上升照理來說會改善miss rate。這邊看不出來應該是cache size：4K已經足夠大，再往上提升也不會有改善。

  ▼ **Source Code**

```
#include <iostream>
#include <cstdio>
#include<iomanip>
#include <math.h>
using namespace std;

struct cache_content {
    bool v;
    unsigned int tag;
};

const int K = 1024;
```

```
void simulate(int cacheSizeKB, int block_size, string cacheName)
{
    unsigned int cacheSizeByte = cacheSizeKB * K, tag, index, byteAddress;
    int accessNum = 0, missNum = 0, hitNum = 0;
    string fileName = cacheName + ".txt";

    int offset_bit = (int)log2(block_size);
    int index_bit = (int)log2(cacheSizeByte / block_size);
    int line = cacheSizeByte >> (offset_bit);   //block number(line) = cache size / block size

    cache_content* cache = new cache_content[line];

    for (int j = 0; j < line; j++)
    {
        cache[j].v = false;
    }

    FILE* fp = fopen(fileName.c_str(), "r");  // FILE* fp = fopen("DCACHE.txt", "r");
    if (!fp)
    {
        cerr << "test file open error!" << endl;
    }

    while (fscanf(fp, "%x", &byteAddress) != EOF)  //Define the type of input data : Hexadecimal Integer(%x)
    {
        accessNum++;
        index = (byteAddress >> offset_bit) & (line - 1);
        tag = byteAddress >> (index_bit + offset_bit);

        if (cache[index].v && cache[index].tag == tag)  //cache hit
        {
            cache[index].v = true;
            hitNum++;
        }
        else                                            //cachemiss
        {
            cache[index].v = true;
            cache[index].tag = tag;
            missNum++;
        }
    }

    fclose(fp);

    delete[] cache;

    cout << cacheName << endl;
    cout << "Cache_size:" << cacheSizeKB << "K" << endl;
    cout << "Block_size:" << block_size << endl;
    cout << "Hit rate: " << fixed << setprecision(2) << (hitNum / (double)accessNum * 100) << "%";
    cout << " (" << hitNum << "),  ";
    cout << "Miss rate: " << fixed << setprecision(2) << (missNum / (double)accessNum * 100) << "%";
    cout << " (" << missNum << ")" << endl << endl;
}


int main(void)
{
    simulate(4, 16, "ICACHE");
    simulate(4, 32, "ICACHE");
    simulate(4, 64, "ICACHE");
    simulate(4, 128, "ICACHE");
    simulate(4, 256, "ICACHE");

    simulate(16, 16, "ICACHE");
    simulate(16, 32, "ICACHE");
    simulate(16, 64, "ICACHE");
    simulate(16, 128, "ICACHE");
    simulate(16, 256, "ICACHE");

    simulate(64, 16, "ICACHE");
    simulate(64, 32, "ICACHE");
    simulate(64, 64, "ICACHE");
    simulate(64, 128, "ICACHE");
    simulate(64, 256, "ICACHE");

    simulate(256, 16, "ICACHE");
    simulate(256, 32, "ICACHE");
    simulate(256, 64, "ICACHE");
    simulate(256, 128, "ICACHE");
    simulate(256, 256, "ICACHE");
```

```
    simulate(4, 16, "DCACHE");
    simulate(4, 32, "DCACHE");
    simulate(4, 64, "DCACHE");
    simulate(4, 128, "DCACHE");
    simulate(4, 256, "DCACHE");

    simulate(16, 16, "DCACHE");
    simulate(16, 32, "DCACHE");
    simulate(16, 64, "DCACHE");
    simulate(16, 128, "DCACHE");
    simulate(16, 256, "DCACHE");

    simulate(64, 16, "DCACHE");
    simulate(64, 32, "DCACHE");
    simulate(64, 64, "DCACHE");
    simulate(64, 128, "DCACHE");
    simulate(64, 256, "DCACHE");

    simulate(256, 16, "DCACHE");
    simulate(256, 32, "DCACHE");
    simulate(256, 64, "DCACHE");
    simulate(256, 128, "DCACHE");
    simulate(256, 256, "DCACHE");

    return 0;
}
```

▼ **Console Output**

```
ICACHE
Cache_size:4K
Block_size:16
Hit rate: 97.83% (721),  Miss rate: 2.17% (16)

ICACHE
Cache_size:4K
Block_size:32
Hit rate: 98.91% (729),  Miss rate: 1.09% (8)

ICACHE
Cache_size:4K
Block_size:64
Hit rate: 99.46% (733),  Miss rate: 0.54% (4)

ICACHE
Cache_size:4K
Block_size:128
Hit rate: 99.73% (735),  Miss rate: 0.27% (2)

ICACHE
Cache_size:4K
Block_size:256
Hit rate: 99.86% (736),  Miss rate: 0.14% (1)

ICACHE
Cache_size:16K
Block_size:16
Hit rate: 97.83% (721),  Miss rate: 2.17% (16)

ICACHE
Cache_size:16K
Block_size:32
Hit rate: 98.91% (729),  Miss rate: 1.09% (8)

ICACHE
Cache_size:16K
Block_size:64
Hit rate: 99.46% (733),  Miss rate: 0.54% (4)

ICACHE
Cache_size:16K
Block_size:128
Hit rate: 99.73% (735),  Miss rate: 0.27% (2)

ICACHE
Cache_size:16K
Block_size:256
Hit rate: 99.86% (736),  Miss rate: 0.14% (1)
```

```
ICACHE
Cache_size:64K
Block_size:16
Hit rate: 97.83% (721),  Miss rate: 2.17% (16)

ICACHE
Cache_size:64K
Block_size:32
Hit rate: 98.91% (729),  Miss rate: 1.09% (8)

ICACHE
Cache_size:64K
Block_size:64
Hit rate: 99.46% (733),  Miss rate: 0.54% (4)

ICACHE
Cache_size:64K
Block_size:128
Hit rate: 99.73% (735),  Miss rate: 0.27% (2)

ICACHE
Cache_size:64K
Block_size:256
Hit rate: 99.86% (736),  Miss rate: 0.14% (1)

ICACHE
Cache_size:256K
Block_size:16
Hit rate: 97.83% (721),  Miss rate: 2.17% (16)

ICACHE
Cache_size:256K
Block_size:32
Hit rate: 98.91% (729),  Miss rate: 1.09% (8)

ICACHE
Cache_size:256K
Block_size:64
Hit rate: 99.46% (733),  Miss rate: 0.54% (4)

ICACHE
Cache_size:256K
Block_size:128
Hit rate: 99.73% (735),  Miss rate: 0.27% (2)

ICACHE
Cache_size:256K
Block_size:256
Hit rate: 99.86% (736),  Miss rate: 0.14% (1)

DCACHE
Cache_size:4K
Block_size:16
Hit rate: 94.44% (119),  Miss rate: 5.56% (7)

DCACHE
Cache_size:4K
Block_size:32
Hit rate: 96.83% (122),  Miss rate: 3.17% (4)

DCACHE
Cache_size:4K
Block_size:64
Hit rate: 98.41% (124),  Miss rate: 1.59% (2)

DCACHE
Cache_size:4K
Block_size:128
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)

DCACHE
Cache_size:4K
Block_size:256
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)

DCACHE
Cache_size:16K
Block_size:16
Hit rate: 94.44% (119),  Miss rate: 5.56% (7)

DCACHE
```

```
Cache_size:16K
Block_size:32
Hit rate: 96.83% (122),  Miss rate: 3.17% (4)

DCACHE
Cache_size:16K
Block_size:64
Hit rate: 98.41% (124),  Miss rate: 1.59% (2)

DCACHE
Cache_size:16K
Block_size:128
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)

DCACHE
Cache_size:16K
Block_size:256
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)

DCACHE
Cache_size:64K
Block_size:16
Hit rate: 94.44% (119),  Miss rate: 5.56% (7)

DCACHE
Cache_size:64K
Block_size:32
Hit rate: 96.83% (122),  Miss rate: 3.17% (4)

DCACHE
Cache_size:64K
Block_size:64
Hit rate: 98.41% (124),  Miss rate: 1.59% (2)

DCACHE
Cache_size:64K
Block_size:128
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)

DCACHE
Cache_size:64K
Block_size:256
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)

DCACHE
Cache_size:256K
Block_size:16
Hit rate: 94.44% (119),  Miss rate: 5.56% (7)

DCACHE
Cache_size:256K
Block_size:32
Hit rate: 96.83% (122),  Miss rate: 3.17% (4)

DCACHE
Cache_size:256K
Block_size:64
Hit rate: 98.41% (124),  Miss rate: 1.59% (2)

DCACHE
Cache_size:256K
Block_size:128
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)

DCACHE
Cache_size:256K
Block_size:256
Hit rate: 99.21% (125),  Miss rate: 0.79% (1)
```

- **set_associative_cache**

  ▼ **Description**

  首先，先將資料處理成有用的樣子

```
int cacheSizeByte = cacheSizeKB * K;
int set_count = (int)(cacheSizeByte / blockSize / Associate);
```

```
int offset_bit = (int)log2(blockSize);
int set_index_bit = (int)log2(cacheSizeByte / blockSize / Associate);
```

其中set_count 代表 #set，所以比起Direct Mapped Cache 的index，要多/ Associate

而offset_bit, set_index_bit 則是用<math.h> 中的log2() 來找出#bits

接下來，File I/O 用 C Library (<cstdio>)中定義的function 來處理 得到 LRU.txt 這個 file 的 head。

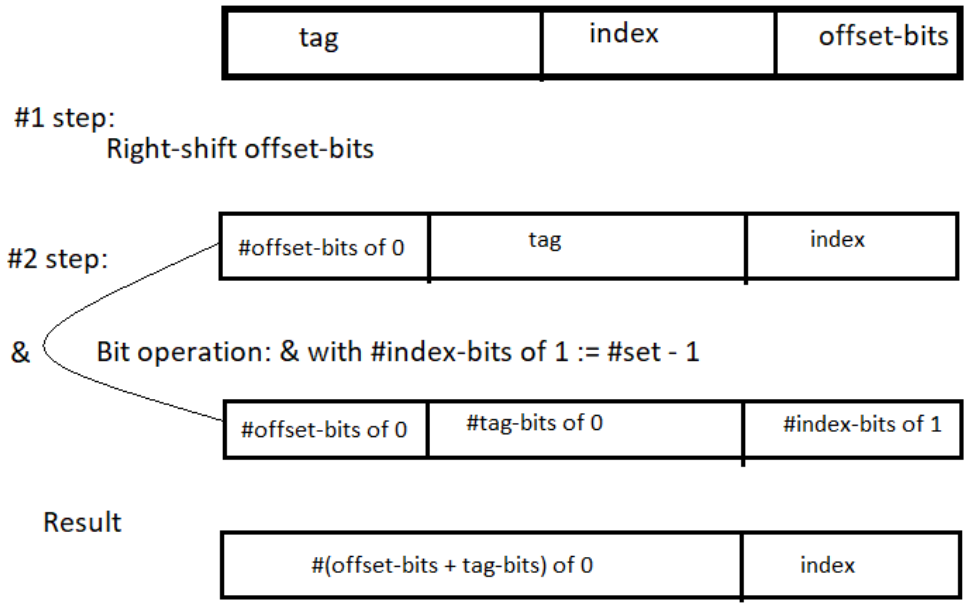
```
FILE *fp = fopen("LRU.txt", "r");
   if (!fp)
   {
       cerr << "test file open error!" << endl;
   }
```

並使用fscanf(FILE *stream, const char *format, ...) 來讀取每一行 LRU.txt 的內容(Byte Address) 至變數 byteAddress 中

```
while (fscanf(fp, "%x", &byteAddress) != EOF)  //Define the type of input data : Hexadecimal Integer(%x)
{
// access count++
// LRU Replacement Algorithm
}
```

LRU Replacement Algorithm

這次Set Associativie LRU Cache 以 一個 vector of list 來實作，其中 vector 代表所有 set，而 list 則是代表了其中的一個 set 並以push back 和 pop front 來實做出 queue 的效果。 在開始 LRU Replacement 判斷之前我們需要先得到input address 的 index 跟 tag。 index 的作法是利用 (byteAddress >> offset_bit) & (set_count - 1) 來 fiter 出 index，想法如下。 tag 則是用 byteAddress >> (set_index_bit + offset_bit) 直接獲得。



而 LRU Replacement Algorithm 的邏輯為, 利用 STL 中 <algorithm> 的 find(start, end, target) 得到的 pointer 來判斷有沒有 hit

```
list<int>::iterator findIter = find(LRU_cache[set_index].begin(), LRU_cache[set_index].end(), tag);
```

有的話便 hit count++ 並將這個tag 移至 list 的最尾端，變最新使用過的tag;

若沒有的話 miss count++，再看該 set 是否已滿，滿的話就pop front 。並把 tag push back 到 list 的尾端

以上便是 LRU Replacement Algorithm 的實做。

最後，走完整個while loop 變得到 access_count, hit_count, miss_count 可以算結果了。

▼ **Discussion**

這次在同一個 Cache Size 中基本符合 set associativity 上升 miss rate 下降的趨勢，唯獨在 I KB Cache中 4 way performance 比 8 way 好，想了很久猜測是在這次的測資中，4 way 的set index 分配到的input 個數比較均勻 miss rate 因此比較低 ，儘管 8 way inclusive of 4 way 但這前提是進到4 way 的同個index的 input 也要 map 到 同個 8 way 的index 才對。

而Cache size 上升， miss rate 下降的趨勢則和上課所教的一致，唯獨在32 K 和 64K high asscociativity的地方表現一樣，應該是在這次測資中 index number的分布在 32K 與 64K 是一致的。

▼ **Source Code**

```
#include <iostream>
#include <fstream>
#include <cstdio>
#include <iomanip>
#include <math.h>
#include <string>
#include <list>
#include <vector>
#include <algorithm>
#define K 1024
using namespace std;

void simulate(int cacheSizeKB, int blockSize, int Associate) {

    int cacheSizeByte = cacheSizeKB * K;
    int set_count = (int)(cacheSizeByte / blockSize / Associate);
    int miss_count = 0, hit_count = 0, access_count = 0;
    int byteAddress, tag, set_index;
    // Init LRU Size
    vector< list<int> > LRU_cache(set_count);

    int offset_bit = (int)log2(blockSize);
    int set_index_bit = (int)log2(cacheSizeByte / blockSize / Associate);

    FILE *fp = fopen("LRU.txt", "r");
    if (!fp)
    {
        cerr << "test file open error!" << endl;
    }

    while (fscanf(fp, "%x", &byteAddress) != EOF)  //Define the type of input data : Hexadecimal Integer(%x)
    {
        // bit extraction
        set_index = (byteAddress >> offset_bit) & (set_count - 1);
        tag = byteAddress >> (set_index_bit + offset_bit);

        list<int>::iterator findIter = find(LRU_cache[set_index].begin(), LRU_cache[set_index].end(), tag);

        access_count++;

        if (findIter != LRU_cache[set_index].end())  //cache hit
        {
            hit_count++;
            LRU_cache[set_index].remove(tag);
            LRU_cache[set_index].push_back(tag);
        }
        else  //cachemiss
        {
            miss_count++;

            if(LRU_cache[set_index].size() == Associate) {
                LRU_cache[set_index].pop_front();
```

```
            }

            LRU_cache[set_index].push_back(tag);

        }
    }

    fclose(fp);
    cout<< Associate << "-N Way" << endl;
    cout << "Cache_size:" << cacheSizeKB << "K" << endl;
    cout << "Block_size:" << blockSize << endl;
    cout << "Hit rate: " << fixed << setprecision(2) << (hit_count / (double)access_count * 100) << "%";
    cout << " (" << hit_count << "),  ";
    cout << "Miss rate: " << fixed << setprecision(2) << (miss_count / (double)access_count * 100) << "%";
    cout << " (" << miss_count << ")"<<endl<<endl;

    // ofstream outFile("Result.txt", ios::out);
    // outFile<< Associate << "-N Way" << endl;
    // outFile << "Cache_size:" << cacheSizeKB << "K" << endl;
    // outFile << "Block_size:" << blockSize << endl;
    // outFile << "Hit rate: " << fixed << setprecision(2) << (hit_count / (double)access_count * 100) << "%";
    // outFile << " (" << hit_count << "),  ";
    // outFile << "Miss rate: " << fixed << setprecision(2) << (miss_count / (double)access_count * 100) << "%";
    // outFile << " (" << miss_count << ")"<<endl<<endl;

}

int main(void)
{
    simulate(1, 64, 1);
    simulate(1, 64, 2);
    simulate(1, 64, 4);
    simulate(1, 64, 8);

    simulate(2, 64, 1);
    simulate(2, 64, 2);
    simulate(2, 64, 4);
    simulate(2, 64, 8);

    simulate(4, 64, 1);
    simulate(4, 64, 2);
    simulate(4, 64, 4);
    simulate(4, 64, 8);

    simulate(8, 64, 1);
    simulate(8, 64, 2);
    simulate(8, 64, 4);
    simulate(8, 64, 8);

    simulate(16, 64, 1);
    simulate(16, 64, 2);
    simulate(16, 64, 4);
    simulate(16, 64, 8);

    simulate(32, 64, 1);
    simulate(32, 64, 2);
    simulate(32, 64, 4);
    simulate(32, 64, 8);

    simulate(64, 64,1);
    simulate(64, 64,2);
    simulate(64, 64,4);
    simulate(64, 64,8);

    simulate(128, 64,1);
    simulate(128, 64,2);
    simulate(128, 64,4);
    simulate(128, 64,8);

    return 0;
}
```

▼ **Console Output**

1-N Way

Cache_size:1K

Block_size:64

Hit rate: 88.93% (5737), Miss rate: 11.07% (714)

2-N Way

Cache_size:1K

Block_size:64

Hit rate: 91.64% (5912), Miss rate: 8.36% (539)

4-N Way

Cache_size:1K

Block_size:64

Hit rate: 92.22% (5949), Miss rate: 7.78% (502)

8-N Way

Cache_size:1K

Block_size:64

Hit rate: 92.17% (5946), Miss rate: 7.83% (505)

1-N Way

Cache_size:2K

Block_size:64

Hit rate: 91.72% (5917), Miss rate: 8.28% (534)

2-N Way

Cache_size:2K

Block_size:64

Hit rate: 94.82% (6117), Miss rate: 5.18% (334)

4-N Way

Cache_size:2K

Block_size:64

Hit rate: 95.81% (6181), Miss rate: 4.19% (270)

8-N Way

Cache_size:2K

Block_size:64

Hit rate: 96.02% (6194), Miss rate: 3.98% (257)

1-N Way

Cache_size:4K

Block_size:64

Hit rate: 94.53% (6098), Miss rate: 5.47% (353)

2-N Way

Cache_size:4K

Block_size:64

Hit rate: 96.37% (6217), Miss rate: 3.63% (234)

4-N Way

Cache_size:4K

Block_size:64

Hit rate: 96.93% (6253), Miss rate: 3.07% (198)

8-N Way

Cache_size:4K

Block_size:64

Hit rate: 97.19% (6270), Miss rate: 2.81% (181)

1-N Way

Cache_size:8K

Block_size:64

Hit rate: 95.97% (6191), Miss rate: 4.03% (260)

2-N Way

Cache_size:8K

Block_size:64

Hit rate: 97.02% (6259), Miss rate: 2.98% (192)

4-N Way

Cache_size:8K

Block_size:64

Hit rate: 97.33% (6279), Miss rate: 2.67% (172)

8-N Way

Cache_size:8K

Block_size:64

Hit rate: 97.55% (6293), Miss rate: 2.45% (158)

1-N Way

Cache_size:16K

Block_size:64

Hit rate: 96.84% (6247), Miss rate: 3.16% (204)

2-N Way

Cache_size:16K

Block_size:64

Hit rate: 97.63% (6298), Miss rate: 2.37% (153)

4-N Way

Cache_size:16K

Block_size:64

Hit rate: 97.66% (6300), Miss rate: 2.34% (151)

8-N Way

Cache_size:16K

Block_size:64

Hit rate: 97.71% (6303), Miss rate: 2.29% (148)

1-N Way

Cache_size:32K

Block_size:64

Hit rate: 97.46% (6287), Miss rate: 2.54% (164)

2-N Way

Cache_size:32K

Block_size:64

Hit rate: 97.67% (6301), Miss rate: 2.33% (150)

4-N Way

Cache_size:32K

Block_size:64

Hit rate: 97.72% (6304), Miss rate: 2.28% (147)

8-N Way

Cache_size:32K

Block_size:64

Hit rate: 97.72% (6304), Miss rate: 2.28% (147)

1-N Way

Cache_size:64K

Block_size:64

Hit rate: 97.66% (6300), Miss rate: 2.34% (151)

2-N Way

Cache_size:64K

Block_size:64

Hit rate: 97.71% (6303), Miss rate: 2.29% (148)

4-N Way

Cache_size:64K

Block_size:64

Hit rate: 97.72% (6304), Miss rate: 2.28% (147)

8-N Way

Cache_size:64K

Block_size:64

Hit rate: 97.72% (6304), Miss rate: 2.28% (147)

1-N Way

Cache_size:128K

Block_size:64

Hit rate: 97.67% (6301), Miss rate: 2.33% (150)

2-N Way

Cache_size:128K

Block_size:64

Hit rate: 97.72% (6304), Miss rate: 2.28% (147)

4-N Way

Cache_size:128K

Block_size:64

Hit rate: 97.72% (6304), Miss rate: 2.28% (147)

8-N Way

Cache_size:128K

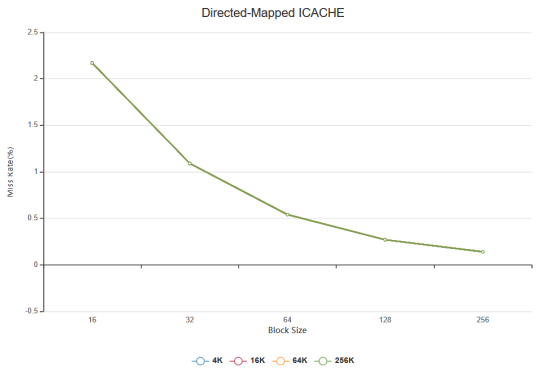
Block_size:64

Hit rate: 97.72% (6304), Miss rate: 2.28% (147)

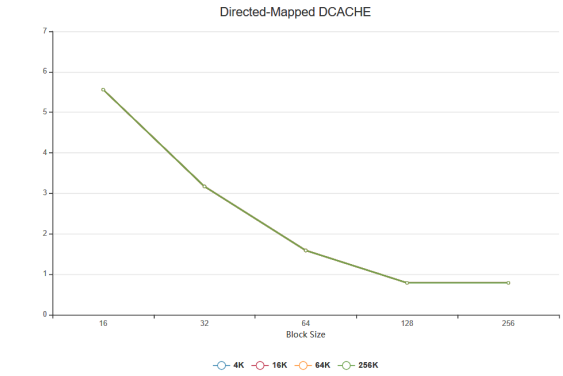
Implementation results

- Basic Problem

▼ I-CACHE



▼ D-CACHE



I CACHE

Aa Block Size	# 4K	# 16K	# 64K	# 256K
16	2.17	2.17	2.17	2.17
32	1.09	1.09	1.09	1.09
64	0.54	0.54	0.54	0.54
128	0.27	0.27	0.27	0.27
256	0.14	0.14	0.14	0.14

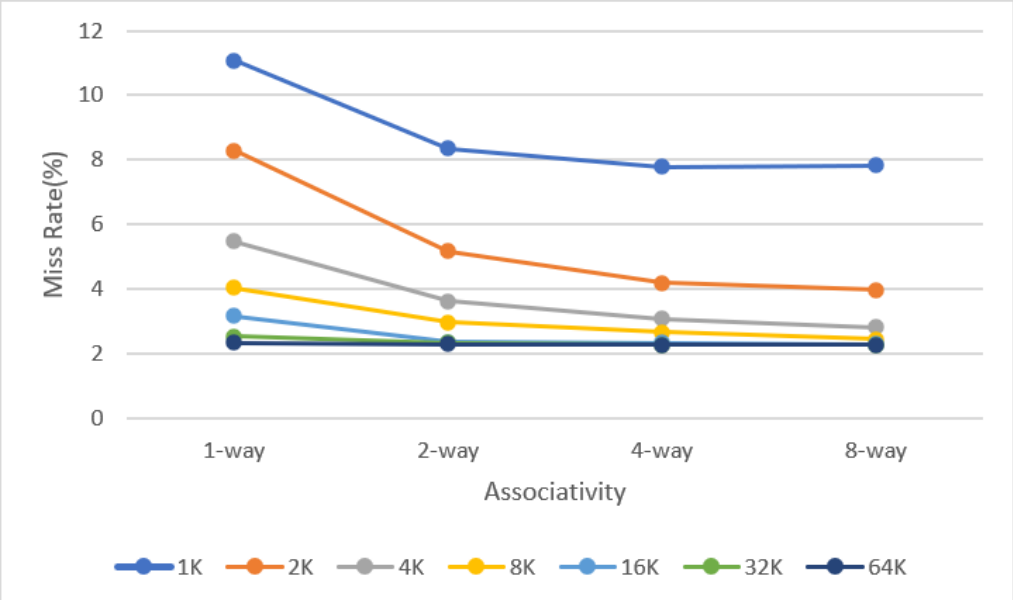
D CACHE

Aa Block Size	# 4K	# 16K	# 64K	# 256K
16	5.56	5.56	5.56	5.56
32	3.17	3.17	3.17	3.17
64	1.59	1.59	1.59	1.59
128	0.79	0.79	0.79	0.79
256	0.79	0.79	0.79	0.79

- Advanced Problem

▼ **Set Associativity Cache**

	1-way	2-way	4-way	8-way
1K	Hit rate: 88.93% Miss rate: 11.07%	Hit rate: 91.64% Miss rate: 8.36%	Hit rate: 92.22% Miss rate: 7.78%	Hit rate: 92.17% Miss rate: 7.83%
2K	Hit rate: 91.72% Miss rate: 8.28%	Hit rate: 94.82% Miss rate: 5.18%	Hit rate: 95.81% Miss rate: 4.19%	Hit rate: 96.02% Miss rate: 3.98%
4K	Hit rate: 94.53% Miss rate: 5.47%	Hit rate: 96.37% Miss rate: 3.63%	Hit rate: 96.93% Miss rate: 3.07%	Hit rate: 97.19% Miss rate: 2.81%
8K	Hit rate: 95.97% Miss rate: 4.03%	Hit rate: 97.02% Miss rate: 2.98%	Hit rate: 97.33% Miss rate: 2.67%	Hit rate: 97.55% Miss rate: 2.45%
16K	Hit rate: 96.84% Miss rate: 3.16%	Hit rate: 97.63% Miss rate: 2.37%	Hit rate: 97.66% Miss rate: 2.34%	Hit rate: 97.71% Miss rate: 2.29%
32K	Hit rate: 97.46% Miss rate: 2.54%	Hit rate: 97.67% Miss rate: 2.33%	Hit rate: 97.72% Miss rate: 2.28%	Hit rate: 97.72% Miss rate: 2.28%



Problems encountered and solutions

這次在 File I/O 與 bit operation 還有 heximal to decimal, heximal to binary 之間的處理花滿多心思去選擇的，原本嘗試用自己寫的function ，但很容易寫錯，才發現STL有很多方便的function ，使用起來 code 變得乾淨許多也比較不易出錯，是比較正確的選擇。

Comment