Computer Organization Lab4

前提摘要:

我們發現此份作業在不同作業系統下會產生不同數據
Trace過後發現為助教提供的此段程式碼造成的
5.9999…在Windows下回傳後接應變數值為5
在Linux下接應變數值卻為6
進而造成Miss數量不同所以數據產生差異

```
1  double log2(double n)
2  {
3      return log(n) / log(double(2));
4  }
```

最後考量助教應該會在Linux下做測試
因此以下數據皆使用Linux測試的數據
版本為 **Ubuntu 18.04**

Basic Problem

ICACHE.txt

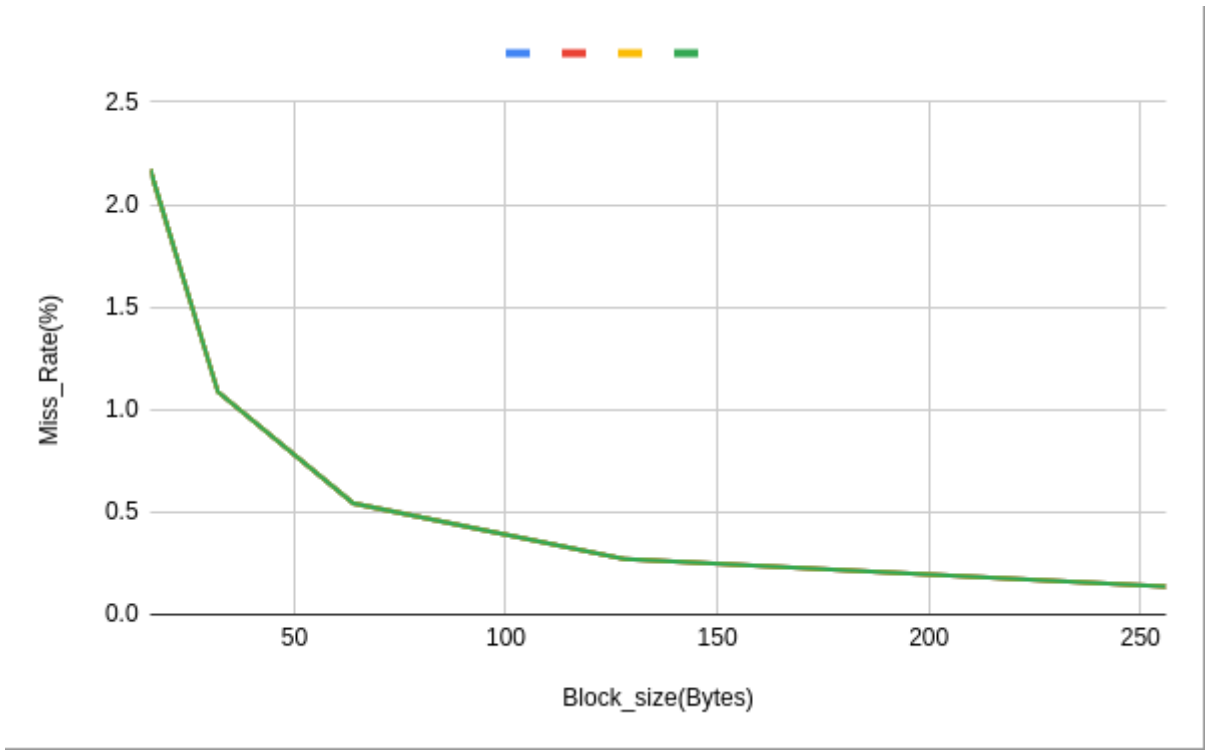
cache_size	block_size	miss_rate
4	16	2.170963365
4	32	1.085481682
4	64	0.5427408412
4	128	0.2713704206
4	256	0.135682103

cache_size	block_size	miss_rate
16	16	2.170963365
16	32	1.085481682
16	64	0.5427408412
16	128	0.2713704206
16	256	0.1356852103

cache_size	block_size	miss_rate
64	16	2.170963365
64	32	1.085481682
64	64	0.5427408412
64	128	0.2713704206
64	256	0.1356852103

cache_size	block_size	miss_rate
256	16	2.170963365
256	32	1.085481682
256	64	0.5427408412
256	128	0.2713704206
256	256	0.1356852103

將結果製作成圖表:



DCACHE.txt

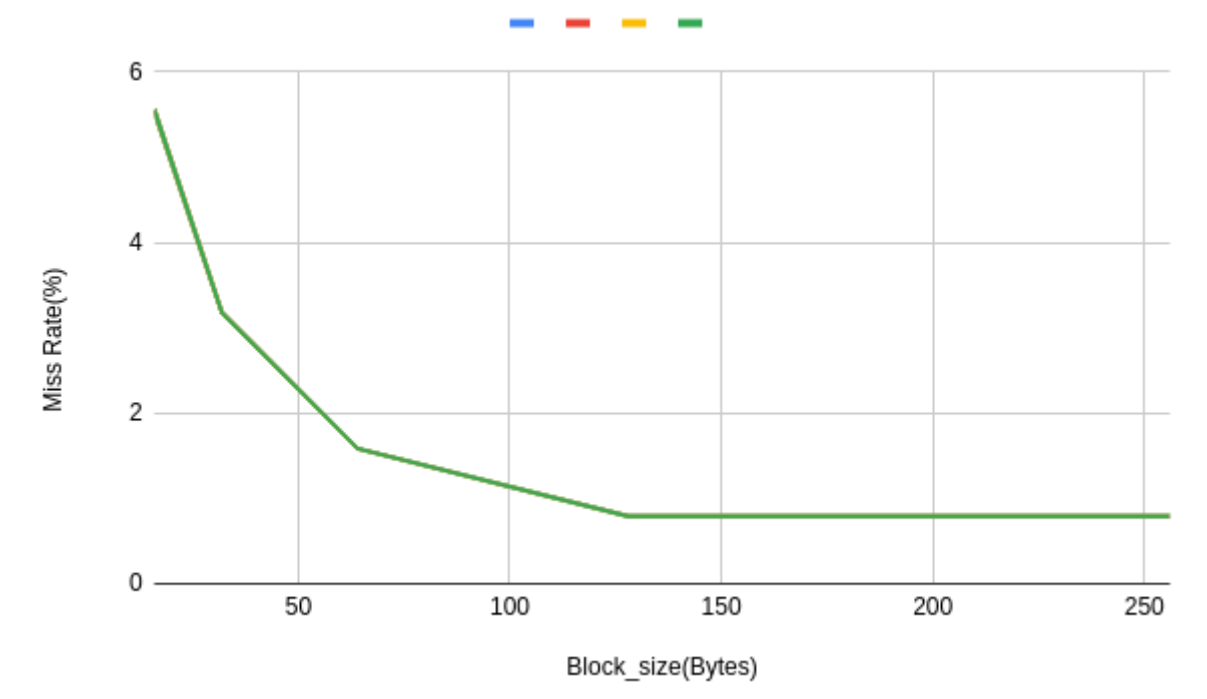
cache_size	block_size	miss_rate
4	16	5.555555556
4	32	3.174603175
4	64	1.587301587
4	128	0.7936507937
4	256	0.7936507937

cache_size	block_size	miss_rate
16	16	5.555555556
16	32	3.174603175
16	64	1.587301587
16	128	0.7936507937
16	256	0.7936507937

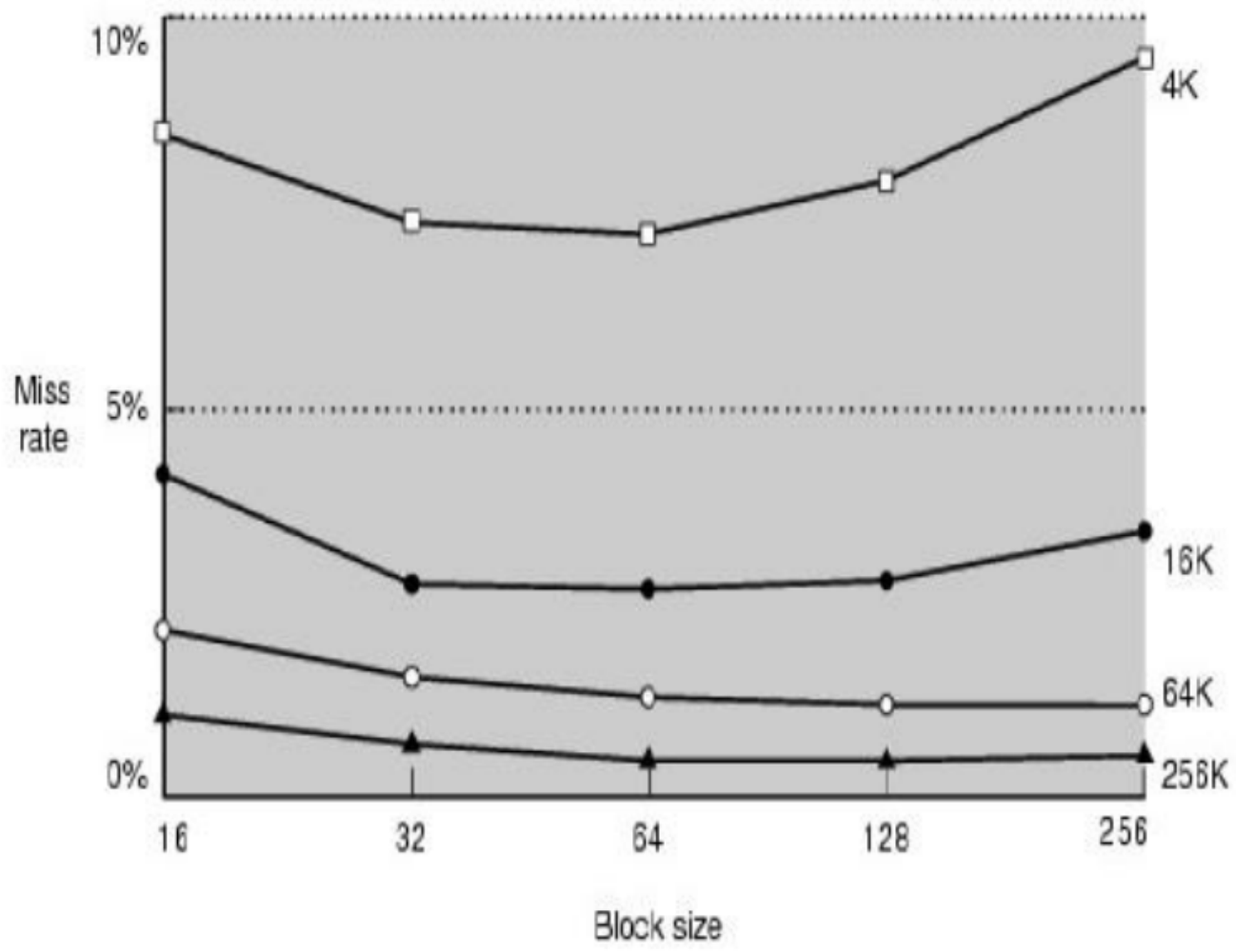
cache_size	block_size	miss_rate
64	16	5.555555556
64	32	3.174603175
64	64	1.587301587
64	128	0.7936507937
64	256	0.7936507937

cache_size	block_size	miss_rate
256	16	5.555555556
256	32	3.174603175
256	64	1.587301587
256	128	0.7936507937
256	256	0.7936507937

將結果製作成圖表:



從圖表中我們可以看到在固定Cache_size下隨著Block_size的增加，Miss Rate會隨之減少，
這是因為較大的Block可以更利用**空間區域性**來達到較低的Miss Rate，
但需要注意的是，
如Spec附的此張圖表，

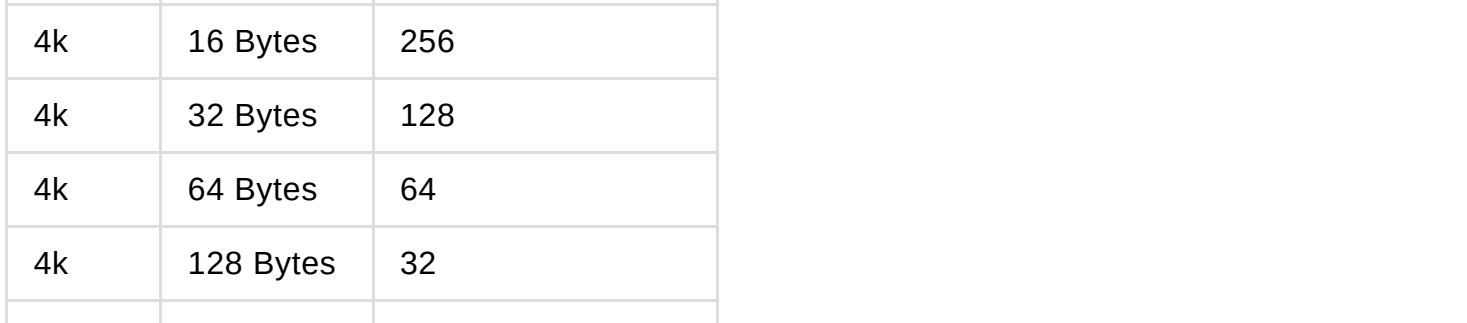


區塊變大時佔快取的比例將會增加，
以4K的Cache為例:

Cache	Block大小	Cache中的Block數
4k	16 Bytes	256
4k	32 Bytes	128
4k	64 Bytes	64
4k	128 Bytes	32
4k	256 Bytes	16

所以如果區塊非常大時快區內的區塊總數下降，
反而**降低空間區域性**，縮小對Miss Rate的好處，
就會發生如上圖Miss Rate後來變高的現象

在此沒有出現此現象的原因:



我們嘗試輸出DCACHE中的命中index與miss的相關數據，
因為DCAHCE檔案本身範圍較小，
可以從圖中看到一開始Cache:4k，Cache中的Block數為256，
但命中的index最大卻只到6相當的小，
因此除了從一開始的部份資料miss後放入cache內，

之後資料幾乎都被幾個Block所包覆，
範圍小，Hit的命中率當然就會很高，
這裡也可以解釋為何就算Block大小從128增大到甚至32768後Miss rate卻沒有任何改變，
因為資料幾乎都被一個Block包住了!

最後附上驗證的截圖，可以看到Miss到後來只出現了僅僅一筆

```
6 2 4 6  miss: 7 5.555555556%
0 1 3 3 1 2 3  miss: 4 3.174603175%
1 1 0 0 1 1 0 1 1  miss: 2 1.587301587%
0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
0 0 0 0 0 0 0 0 0 0 0 0  miss: 1 0.7936507937%
```

不同Cache_size卻產生相同的結果

不管是DCACHE或是ICACHE都發生了這個結果，
就是無論是多大的Cache_size結果都跑出了一樣的數據，
(圖中原本有四種線顏色的，但只有一條線表示完全一模一樣把彼此蓋住了)
我們認為這個發生的原因與上述**數值範圍太小**有關，
在cache_size:256k block_size:64(Bytes)中命中的index最大僅僅為3，
推估ICACHE中最大的數不超過300，
DCACHE更比ICACHE還小，
因此才會發生不同Cache_size卻產生相同的結果。

Advanced Problem

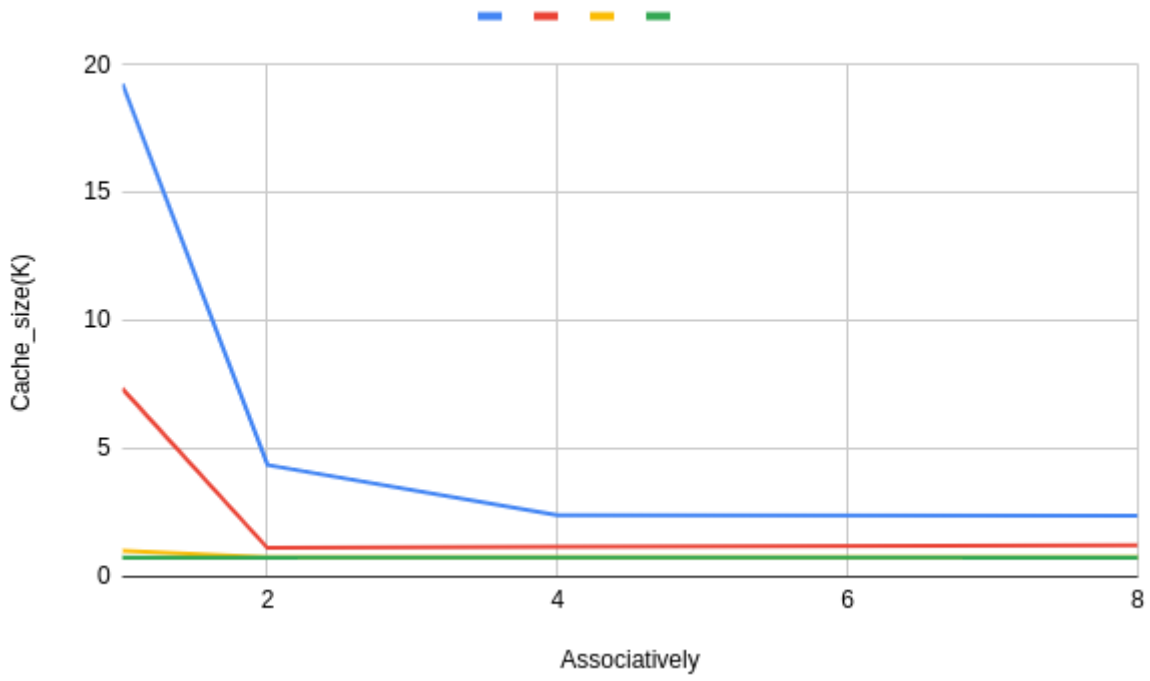
RADIX.txt

Block_Size	Cache_size	Associatively	Miss_Rate
64	4	1	19.25691244
64	4	2	4.366359447
64	4	4	2.396313364
64	4	8	2.374711982

Block_Size	Cache_size	Associatively	Miss_Rate
64	16	1	7.355990783
64	16	2	1.124711982
64	16	4	1.167914747
64	16	8	1.237039171

Block_Size	Cache_size	Associatively	Miss_Rate
64	64	1	1.0109447
64	64	2	0.7531682028
64	64	4	0.7531682028
64	64	8	0.7517281106

Block_Size	Cache_size	Associatively	Miss_Rate
64	256	1	0.7517281106
64	256	2	0.7517281106
64	256	4	0.7517281106
64	256	8	0.7517281106



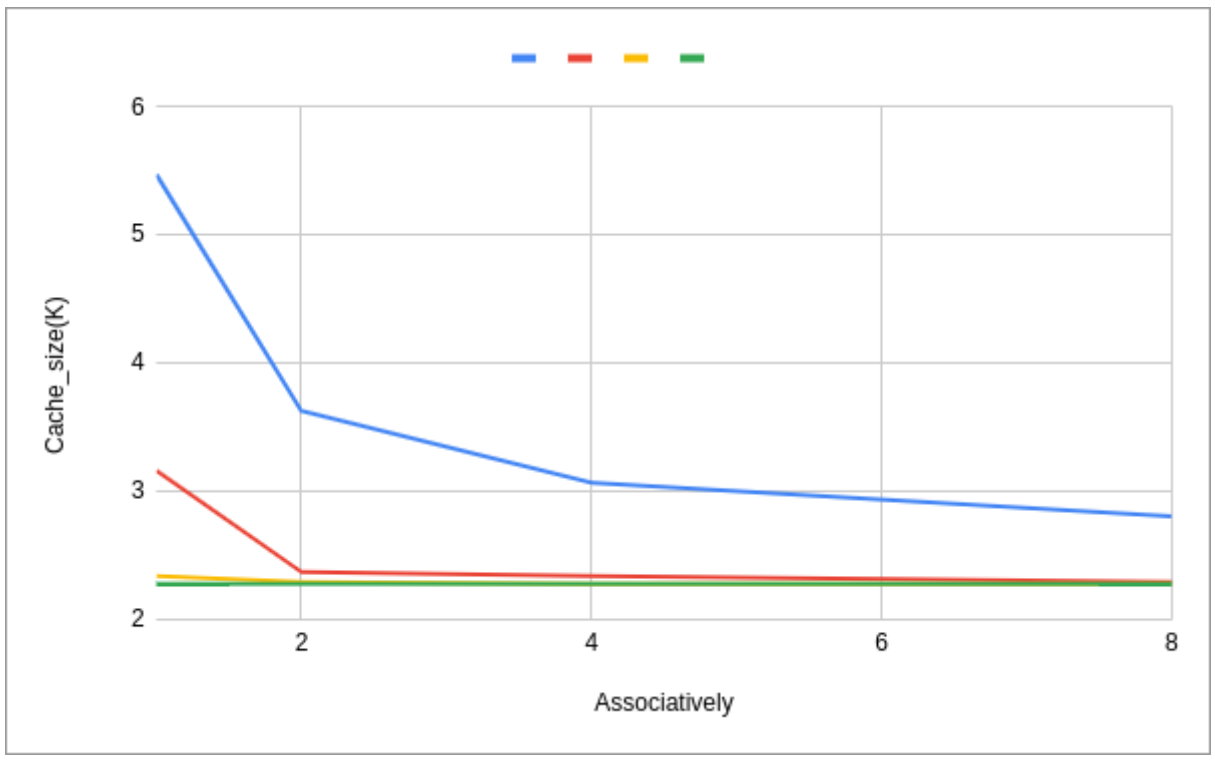
LU.txt

Block_Size	Cache_size	Associatively	Miss_Rate
64	4	1	5.472019842
64	4	2	3.627344598
64	4	4	3.069291583
64	4	8	2.805766548

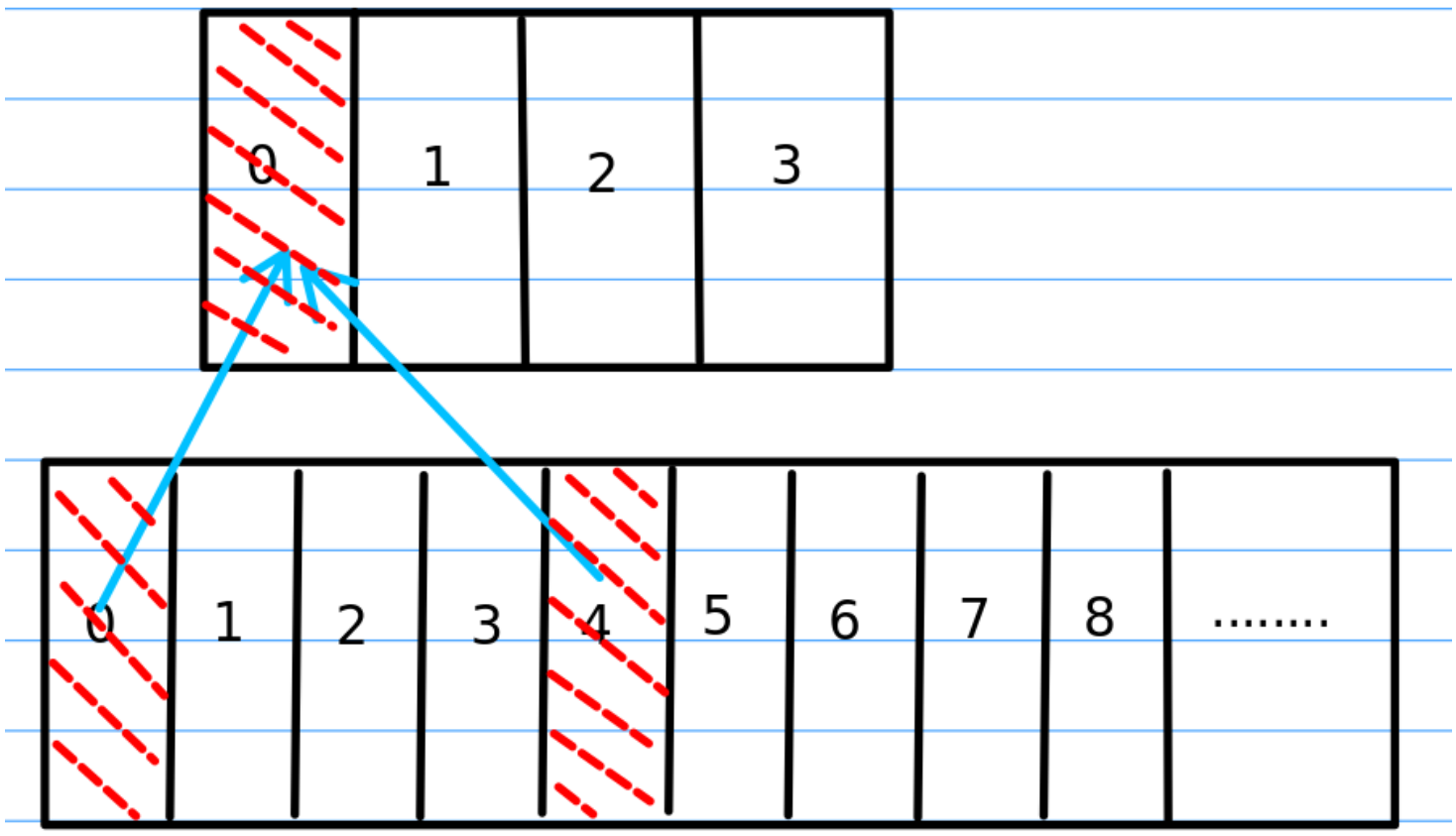
Block_Size	Cache_size	Associatively	Miss_Rate
64	16	1	3.162300419
64	16	2	2.371725314
64	16	4	2.340722369
64	16	8	2.294217951

Block_Size	Cache_size	Associatively	Miss_Rate
64	64	1	2.340722369
64	64	2	2.294217951
64	64	4	2.278716478
64	64	8	2.278716478

Block_Size	Cache_size	Associatively	Miss_Rate
64	256	1	2.278716478
64	256	2	2.278716478
64	256	4	2.278716478
64	256	8	2.278716478

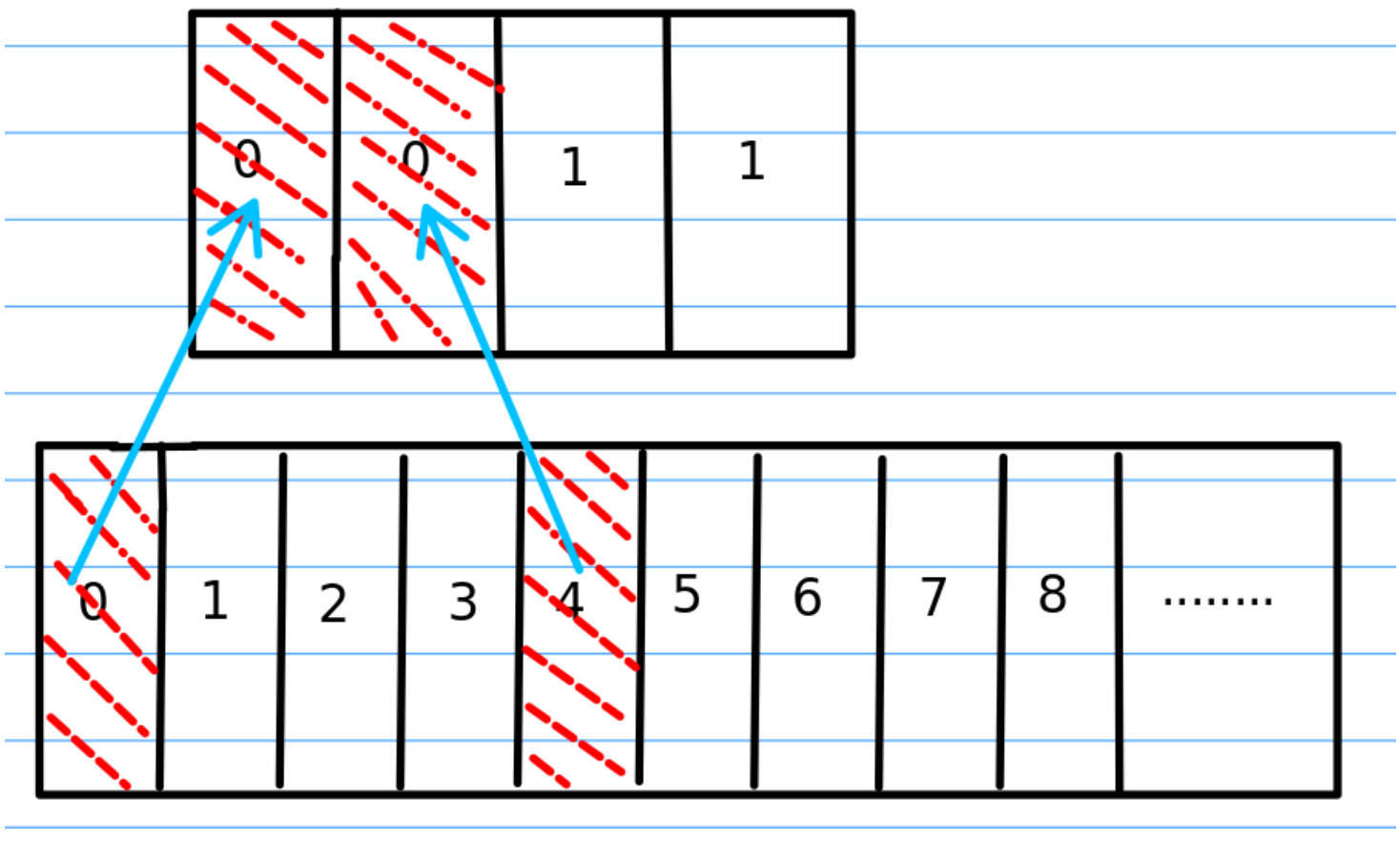


首先，
使用Associative Cache的好處是，
如果使用Direct Mapped Cache，
像是萬一遇到以下這種情況:



Block Address 0與4不斷的搶Cache中的Block0，
一旁的1、2、3卻沒充分利用，
造成Miss Rate大幅增加

改用Associative Cache(此圖為2-way):



在Set中有2個Block可以給相同Set number的Block Address存放，
大幅降低了Miss Rate(代價是增加了Hit Time)

然後如果發生碰撞需要替換的話有兩種方法:

1. 隨機替換
2. LRU:最不常用的被換掉

此處我們根據spec要求實做LRU的方法後得到的數據如上，
我們可以輕易的看到Miss Ratea明顯有降低，
在RADIX.txt中Cache_size=4K下最為明顯的從約19%最後到2.3%，

在Basic中，隨著Cache_size上升結果卻一樣，
但Advanced的這兩個檔案的結果來看也可以看到隨著Cache_size上升Miss Rate有下降的現象，
我認為與Basic中推估的是因為檔案測資過小有相符，
在LU.txt與RADIX.txt中數值大了許多且資料數也多，
為更符合拿來觀察的數據。

Associatively增加Miss_Rate卻不變

在RADIX.txt中可以觀察到在catch_size:256k開始，
增大了Associatively也沒有辦法降低Miss_Rate，
如果再仔細看，
會發現其實Miss_rate在64K 8-way與停止降低的Miss Rate相同

先列出Cache的Set大小觀察

Cache_size	Block_size	Associatively	index bit
256K	64 Bytes	1	12
256K	64 Bytes	2	11
256K	64 Bytes	4	10
256K	64 Bytes	8	9
64K	64 Bytes	8	7

其他Associatively增加Miss_Rate有降低的:

Cache_size	Block_size	Associatively	index bit
4K	64 Bytes	1	6
4K	64 Bytes	2	5
4K	64 Bytes	4	4
4K	64 Bytes	8	3

Cache_size	Block_size	Associatively	index bit
16K	64 Bytes	1	8
16K	64 Bytes	2	7
16K	64 Bytes	4	6
16K	64 Bytes	8	5

Cache_size	Block_size	Associatively	index bit
64K	64 Bytes	1	10
64K	64 Bytes	2	9
64K	64 Bytes	4	8
64K	64 Bytes	8	7

由於Miss_Rate相同表示Miss數量也相同
至於為何Associatively不影響Miss_Rate，
我們推測是因為**index在256K時Associatively為1、2、4、8時重複了多次且他們的tag也全都一樣**，
因此Miss只會發生在index第一次出現時，之後tag、index因為相同就不斷重複一直Hit。

我們可以簡單透過程式驗證:

在Catch_size=256K，Associatively=1時的情況
左邊為index，右邊為tag，可以看到很多相同的情況

```
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110011 10111111100101
111001110011 10111111100101
111001110011 10111111100101
111001110011 10111111100101
111001110011 10111111100101
111001110100 10111111100101
111001110100 10111111100101
001100010101 00001000000011
111001110100 10111111100101
001111100001 00001000000011
001100000000 00001000000011
111001110100 10111111100101
111001110100 10111111100101
111001110100 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110101 10111111100101
111001110110 10111111100101
111001110110 10111111100101
111001101011 10111111100101
111001101011 10111111100101
111001101011 10111111100101
111001101011 10111111100101
```

進一步將index與tag結合起來
並紀錄重複的數量
我們可以發現這邊符合我們的推測
而且總共147筆資料恰好也為Miss的數量

```
001111010011000001000000011 16
110001100000000001000000010 15
111010110010101111111100101 9
010011011011000001000000011 14
111111011011000001000000010 5
001100011100000001000000011 4
111010110011101111111100101 9
111010110100101111111100101 20
010001100000000001000000011 89
000011001010000001000000011 4
010000100001000001000000011 15
001000000000000001000000001 2
110001100010000001000000010 9
001111011111000001000000011 16
001111001110000001000000011 114
000110110110000001000000011 1
010001110010000001000000011 7
111001100111101111111100101 47
111010010011101111111100101 48
111001100110101111111100101 1047
001111010001000001000000011 17
001111010100000001000000011 16
001111010110000001000000011 16
001111011000000001000000011 16
001111011001000001000000011 16
001110100110000001000000011 4
001111011011000001000000011 17
111001100101101111111100101 578
001111011101000001000000011 16
001111011110000001000000011 59
010001110011000001000000011 15
000010000111000001000000011 3
101101100100000001000000010 80
010001110100000001000000011 12
010001110101000001000000011 7
001000000011000001000000001 1
101101101111000001000000010 29
111010010010101111111100101 6
```

將Associatively=2、4、8情況也驗證:

```
110101000001011111111001011 27
011100000100000100000000101 1
011001000000000100000000110 3
000100001110000100000000110 3
110101100101011111111001011 9
110101100111011111111001011 9
110101101001011111111001011 20
110010100111011111111001011 42
110101101101011111111001011 36
000110010100000100000000110 4
100011000010000100000000101 35
110011001111011111111001011 47
100011000000000100000000110 89
100001000010000100000000110 15
011000000000000100000000110 16
011000101000000100000000110 45
011110011010000100000000110 3
011011001010000100000000101 61
011110100000000100000000110 16
110101110011011111111001011 9
011110100010000100000000110 17
011110100100000100000000110 16
110010101001011111111001011 10
011110100110000100000000110 16
011000000010000100000000110 9
011110101010000100000000110 16
011110101110000100000000110 16
111110110110000100000000101 5
011110011110000100000000110 26
011110110100000100000000110 17
011110101000000100000000110 16
011110110110000100000000110 17
011100110110000100000000101 4
011110111010000100000000110 16
011110111100000100000000110 59
110101101111011111111001011 9
000100010000000100000000110 5
011110111000000100000000110 16
100011101010000100000000110 7
```

```
101011001010111111110010111 9
100101111111011111110010111 97
100110011110111111110010111 47
100110001110111111110010111 110
111101111100001000000001100 16
110000011000001000000001100 245
001001101000001000000001101 20
001000100000001000000001100 5
110001110100001000000001100 1
000110000000001000000001011 15
111101100100001000000001100 16
000010011000001000000001101 5
101010000010111111110010111 27
001101101100001000000001101 14
100110110110111111110010111 7
101010000110111111110010111 27
111100010000001000000001100 9
101011011010111111110010111 36
000110000000001000000001101 89
101011100110111111110010111 9
110001010000001000000001100 45
101001001010111111110010111 6
100101001010111111110010111 60
111100110100001000000001100 3
111101000100001000000001100 17
100101010010111111110010111 10
111101001100001000000001100 16
111101010000001000000001100 16
111101011100001000000001100 16
111101101000001000000001100 17
111101101100001000000001100 17
111101110000001000000001100 16
101001001110111111110010111 48
111101111000001000000001100 59
100100110010111111110010111 6
000111001000001000000001101 7
000111001100001000000001101 15
111100111000001000000001100 114
000110001000001000000001011 9
100110100110111111110010111 201
100111011010111111110010111 33
100000001100001000000000100 1
```

```
106    01011100110111111100101111 9
107    00101010100001000000010110 14
108    11101100000001000000011001 16
109    01011001110111111100101111 9
110    00010001100001000000011010 9
111    00010000100001000000011010 15
112    00110011110111111100101111 47
113    10001010000001000000011001 45
114    11101111100001000000011001 16
115    00111001010111111100101111 1
116    11100110100001000000011001 3
117    10000001000001000000011001 28
118    11100111100001000000011001 26
119    00111001000001000000011010 7
120    00101101110111111100101111 28
121    10001010100001000000011001 11
122    00000001100001000000001001 1
123    01011100010111111100101111 18
124    11100111000001000000011001 114
125    00110011010111111100101111 1047
126    11101000000001000000011001 16
127    00110110110111111100101111 7
128    11101000100001000000011001 17
129    00110100000001000000011010 2
130    00111010000001000000011010 12
131    11101110100001000000011001 16
132    11100110000001000000011001 77
133    11101001000001000000011001 16
134    00110001010111111100101111 9
135    00000100100001000000011010 11
136    11101001100001000000011001 16
137    11101011100001000000011001 16
138    10001110100001000000011001 1
139    10000001000001000000011001 9
140    11101101000001000000011001 17
141    00101111010111111100101111 30
142    11101100100001000000011001 16
143    11101110000001000000011001 16
144    11100010000001000000011001 9
145    01001101000001000000011010 20
146    00111010110111111100101111 160
147    01001001010111111100101111 6
```

藉由這些數據我們可以總結，
造成Associatively增加Miss_Rate卻不變的原因的確是因為數據中的index與tag的組合重複性質過高造成的。