CS224
Section No.: 2
Spring 2020
Lab No.: 6
Defne Betül Çiftci/21802635

Part 1

Question 1 –

No.	Cache Size KB	N way cache	Word Size in bits	Block size (no. of words)	No. of Sets	Tag Size in bits	Index Size (Set No.) in bits	Word Block Offset Size in bits[1]	Byte Offset Size in bits[2]	Block Replacement Policy Needed (Yes/No)
1	2	1	32	4	128	18	7	2	2	No
2	2	2	32	4	64	19	6	2	2	Yes
3	2	4	32	8	16	20	4	3	2	Yes
4	2	Full	32	8	1	24	0	3	2	Yes
9	16	1	16	4	2048	15	11	2	1	No
10	16	2	16	4	1024	16	10	2	1	Yes
11	16	4	8	16	256	17	8	4	0	Yes
12	16	Full	8	16	1	25	0	4	0	Yes

[1] **Word Block Offset Size in bits:** Log_2(No. of words in a block)

[2] **Byte Offset Size in bits:** Log_2(No. of bytes in a word)

Main memory size is 0.5 GB = $\frac{(2^{10})^3}{2} = 2^{29}$ bytes -> physical address is 29 bits

Question 2 – a.

Instruction	Iteration No.				
	1	2	3	4	5
lw $t1, 0xA4($0)	Compulsory	Hit	Hit	Hit	Hit
lw $t2, 0xAC($0)	Compulsory	Hit	Hit	Hit	Hit
lw $t3, 0xA8($0)	Compulsory	Hit	Hit	Hit	Hit

b. 4 GB memory => 4*2^30 = 2^32 bytes => 32 bits in physical address

Cache capacity / Block size = 8 / 2 = 4 sets => 2 set bits, 2 byte offset bits and $\log_2(Block\ size) = \log_2(2) = 1$ block offset bit

2 bits(Byte offset) | 2 bits (Set Bit) | 1 bit (Block offset) | $\left(32 - (2+2+1)\right) = 27$ bits Tag size in bits

And V = 1 bit | Tag = 27 bits | data = 32 bits | data = 32 bits => Total cache memory = (4 sets) * (1+27+32+32) = 368 bits

c. 1 equality checker for tag, 1 MUX for selecting word, 1 AND for hit

Question 3 –

Instruction	Iteration No.				
	1	2	3	4	5
lw $t1, 0xA4($0)	Compulsory	Capacity	Capacity	Capacity	Capacity
lw $t2, 0xAC($0)	Compulsory	Capacity	Capacity	Capacity	Capacity
lw $t3, 0xA8($0)	Capacity	Capacity	Capacity	Capacity	Capacity

b. Calculate tag bits -> 0 set bits (1 set), 0 block offset bits (1 block size), byte offset = 2 bits => 30 bits for tag bits (32 – 2 = 30)

1 set -> V bit | used bit | Tag bits | data (32 bits) | V bit | Tag bits | data (32 bits) = > 2*(1+1+30+32)=128 bits in total.

c. 2 equality checkers to check for tag bits (since there are 2 tags in one set)

1 MUX to select the block (2 blocks inside 1 set)

2 AND (connected to both V's and the result of equality checkers) and 1 OR (connected to AND results) for hit

Question 4 – The solution is inside the .txt file uploaded on Unilica.

```
.data
dimension:  .space 4
size:       .space 4
array:              .space 4
stringfirstInput: .asciiz "Enter the matrix size: "
stringsecInput: .asciiz "Enter the row of the number you want to get: "
string3rdInput: .asciiz "Enter the column of the number you want to get: "
firstOutput:      .asciiz "The number on the (row, column) you wanted: "
rowSum:           .space 4
colSum:           .space 4
comma:            .asciiz ", "
ikinokta:   .asciiz ": "
rowSumPrint:      .asciiz "The summation result done through row-major
summation: "
colSumPrint:      .asciiz "The summation result done through column-major
summation: "
press1row:  .asciiz "\nPress 1 to see the result of row-major summation\n"
press1col:  .asciiz "\nPress 1 to see the result of column-major
summation\n"
rowcolchoice:     .asciiz "\nPress 0 for row display, press 1 for column
display "
rowcolwrong:      .asciiz "\nEnter either 0 or 1 please.\n"
whichcol:   .asciiz "\nWhich column do you want to print? "
whichrow:   .asciiz "\nWhich row do you want to print? "
space:            .asciiz " "
col:        .asciiz "column "
row:        .asciiz "row "
.text
main: la $a0, stringfirstInput
      li $v0, 4
      syscall
      li   $v0, 5
      syscall     #gets input for matrix dimension
      sw   $v0, dimension
check:      ble  $v0, 0, main
      addi $sp, $sp, -8
      sw   $s0, ($sp) #s0 is a temp for dimension
      sw   $s1, 4($sp) #s0 is a temp for size
      lw   $s0, dimension

      multu $s0, $s0
      mflo  $s1
```

```
        sll   $s1, $s1, 2
        sw    $s1, size

        li   $v0, 9
        lw   $a0, size
        syscall
        sw   $v0, array

        lw   $t0, dimension #column
        addi $t0, $t0, 1
        lw   $a0, array #address
        li   $s1, 0
        j    addValuesToCol
goBack:
        la $a0, stringsecInput
        li $v0, 4
        syscall
        li   $v0, 5
        syscall
        move $t0, $v0 #row

        la $a0, string3rdInput
        li $v0, 4
        syscall
        li   $v0, 5
        syscall
        move $t1, $v0 #column

        #address is ((column - 1) x N  + (row - 1))*4
        addi $t1, $t1, -1 #column - 1
        lw   $t2, dimension
        mult $t1, $t2
        mflo $t1 #(column - 1) * N
        addi $t0, $t0, -1 #row - 1
        add  $t0, $t0, $t1 #(column - 1) x N  + (row - 1)
        sll  $t0, $t0, 2   #times 4

        lw   $a0, array #we get the array address
        add  $a0, $a0, $t0 #the address of the element wanted
        lw   $t0, ($a0)
        la   $a0, firstOutput
        li   $v0, 4
        syscall
        move $a0, $t0
        li   $v0, 1
        syscall

        la $a0, press1row
        li $v0, 4
        syscall
        li $v0, 5
        syscall
        beq $v0, 1, rowMajorsumm

        la $a0, press1col
        li $v0, 4
        syscall
```

```
      li $v0, 5
      syscall
      beq $v0, 1, colMajorsumm

      j display

rowMajorsumm:
      lw   $a0, array #get the array address
      addi $a0, $a0, -4
      li   $a1, 0
      li   $t0, 0 #row
      addi $t0, $t0, -1
      lw   $t2, dimension
      mul $t3, $t2, 4 #this is how many we will go up every time inside row
major summation
      li   $s0, 0 #for sum
      li   $s1, 0 #for each number
      j    rowMajorLoopUpper

colMajorsumm:
      lw   $a0, array #get the array address
      li   $t0, 0 #col
      addi $t0, $t0, -1
      lw   $t2, dimension
      li   $s0, 0 #for sum
      li   $s1, 0 #for each number
      j    colMajorLoopUpper

goBackToSum:
      sw   $s0, rowSum
      li $a0, 0xA #to get a new line
        li $v0, 11 #syscall 11 prints the lower 8 bits of $a0 as an ascii
character.
        syscall
      la $a0, rowSumPrint
      li $v0, 4
      syscall

      lw $a0, rowSum
      li $v0, 1
      syscall

      la $a0, press1col
      li $v0, 4
      syscall
      li $v0, 5
      syscall
      beq $v0, 1, colMajorsumm

      j display
goBackToColSum:
      sw $s0, colSum
      li $a0, 0xA #to get a new line
        li $v0, 11 #syscall 11 prints the lower 8 bits of $a0 as an ascii
character.
        syscall
      la $a0, colSumPrint
```

```
	li $v0, 4
	syscall

	move $a0, $s0
	li $v0, 1
	syscall

	j display

rowMajorLoopUpper: #burayı tekrar yap
	li   $t1, 0 #column
	addi $t0, $t0, 1
	addi $a0, $a0, 4
	move $a1, $a0

	bne  $t0, $t2, rowMajorLoopInner
	beq  $t0, $t2, goBackToSum
	rowMajorLoopInner:
		lw   $s1, ($a1)
		add  $s0, $s0, $s1
		add  $a1, $a1, $t3
		addi $t1, $t1, 1
		bne  $t1, $t2, rowMajorLoopInner
		beq  $t1, $t2, rowMajorLoopUpper

colMajorLoopUpper: #row keeps repeating, col is increasing
	li   $t1, 0 #row
	addi $t0, $t0, 1
	bne  $t0, $t2, colMajorLoopInner
	beq  $t0, $t2, goBackToColSum
	colMajorLoopInner:
		lw   $s1, ($a0)
		add  $s0, $s0, $s1
		addi $a0, $a0, 4
		addi $t1, $t1, 1
		bne  $t1, $t2, colMajorLoopInner
		beq  $t1, $t2, colMajorLoopUpper

displaywrong:
	la $a0, rowcolwrong
	li $v0, 4
	syscall
display:
	la $a0, rowcolchoice
	li $v0, 4
	syscall
	li $v0, 5
	syscall #0 is row, 1 is column
	lw   $a1, array #get the address
	lw   $t2, dimension
	beq $v0, 0, dispRow
	beq $v0, 1, dispCol
	j displaywrong
dispRow:
	la $a0, whichrow
	li $v0, 4
	syscall
```

```
	li $v0, 5
	syscall
	move $t0, $v0 #t0 is the row count
	#below part prints out: row x : # # # and the hashes will be numbers
	la $a0, row
	li $v0, 4
	syscall
	move $a0, $t0
	li $v0, 1
	syscall
	la $a0, ikinokta
	li $v0, 4
	syscall

	addi $t0, $t0, -1 #(row - 1)
	mul  $t0, $t0, 4  #           * 4
	li   $t1, 0 #a temp to count for the columns
	li   $t3, 0 #another temp for calculating address each time
	#addi $t2, $t2, 1
dispRowLoop:
	addi $t1, $t1, 1
	#calculate address based on column
	addi $t3, $t1, -1 #column - 1
	mul $t3,  $t3, $t2 #        * N
	mul $t3, $t3, 4 #       * 4
	add $t3, $t3, $t0 #          + (row - 1) * 4
	add $t3, $t3, $a1 #          + Address

	lw $a0, ($t3)
	li $v0, 1
	syscall
	la $a0, space
	li $v0, 4
	syscall

	blt $t1, $t2, dispRowLoop
	bge $t1, $t2, end
dispCol:
	la $a0, whichcol
	li $v0, 4
	syscall
	li $v0, 5
	syscall
	move $t0, $v0 #t0 is the column count

	la $a0, col
	li $v0, 4
	syscall
	move $a0, $t0
	li $v0, 1
	syscall
	la $a0, ikinokta
	li $v0, 4
	syscall

	addi $t0, $t0, -1 #(column - 1)
	mul  $t0, $t0, $t2 #         * N
```

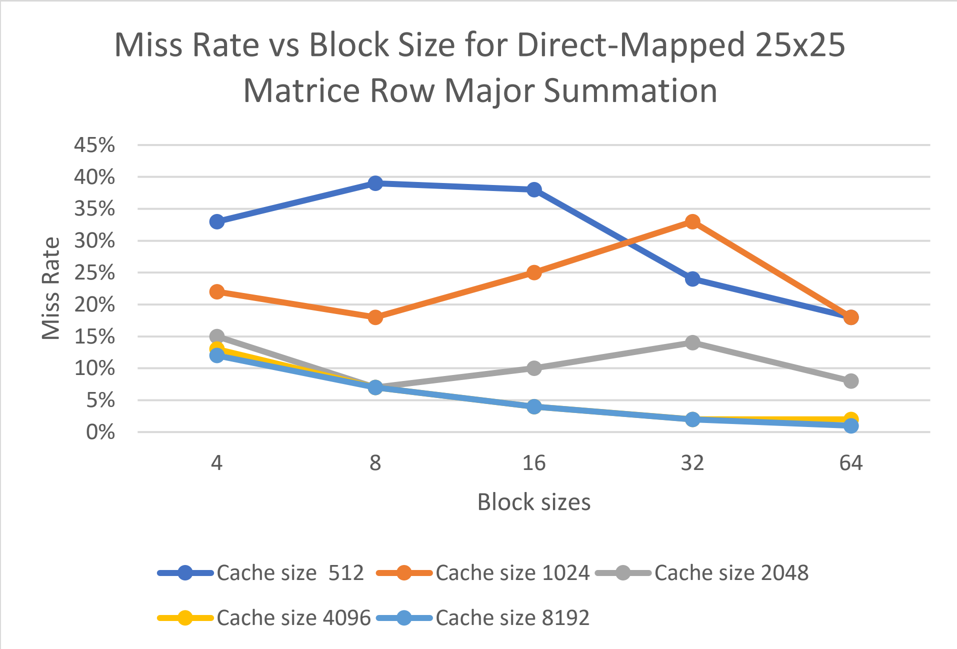
```
        mul $t0, $t0, 4   #            * 4
        add $t0, $a1, $t0 #t0 is now the address to the specific column
dispColLoop:
        lw $s0, ($t0)

        move $a0, $s0
        li $v0, 1
        syscall

        la $a0, space
        li $v0, 4
        syscall

        addi $t0 ,$t0, 4
        addi $t2, $t2, -1 #decrease size temp, increase address
        bne $t2, 0, dispColLoop
        beq $t2, 0, end

calcAddr:
        #address is ((column - 1) x N  + (row - 1))*4
        addi $s1, $s1, -1 #column - 1
        mul $s1, $s1, $t2 #(column - 1) * N
        addi $s0, $s0, -1 #row - 1
        add  $s0, $s0, $s1 #(column - 1) x N  + (row - 1)
        sll  $s0, $s0, 2   #times 4
        jr $ra

end:    lw   $s0, ($sp)
        lw   $s1, ($sp)
        lw   $s2, ($sp)
        addi $sp, $sp, 12
        lw   $s0, ($sp) #s0 is a temp for dimension
        lw   $s1, 4($sp) #s0 is a temp for size
        addi $sp, $sp, 8
        li $v0, 10
        syscall

addValuesToCol:
        addi $t0, $t0, -1
        lw   $t1, dimension #row
        bgt  $t0, 0, addValuesToRow
        beq  $t0, 0, goBack
        addValuesToRow:
              addi $s1, $s1, 1
              sw   $s1, ($a0)
              addi $a0, $a0, 4
              addi $t1, $t1, -1
              bgt  $t1, 0, addValuesToRow
              beq  $t1, 0, addValuesToCol
```

Part 2

a) **Direct Mapped Caches**:

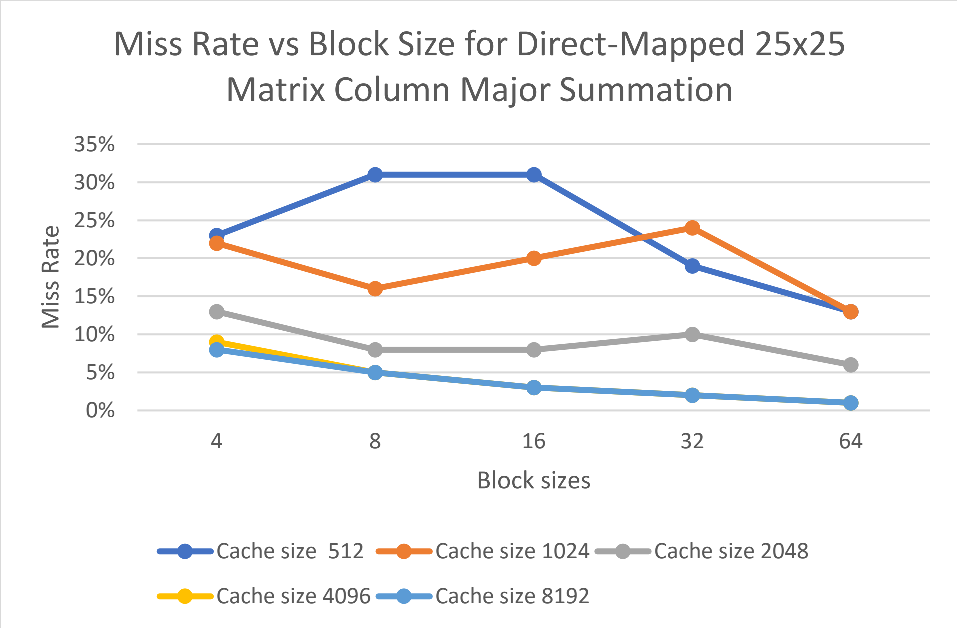
	Block size 4	Block size 8	Block size 16	Block size 32	Block size 64
Cache size 512	33%, 525	39%, 634	38%, 611	24%, 559	18%, 293
Cache size 1024	22%, 352	18%, 286	25%, 580	33%, 530	18%, 284
Cache size 2048	15%, 245	7%, 167	10%, 161	14%, 219	8%, 129
Cache size 4096	13%, 167	7%, 107	4%, 58	2%, 35	2%, 33
Cache size 8192	12%, 140	7%, 107	4%, 58	2%,35	1%, 24

Table for Direct-Mapped 25x25 matrice, Row Major Summation



	Block size 4	Block size 8	Block size 16	Block size 32	Block size 64
Cache size 512	23%, 528	31%, 718	31%, 721	19%, 582	13%, 305
Cache size 1024	22%, 506	16%, 366	20%, 623	24%, 554	13%, 296
Cache size 2048	13%, 306	8%, 134	8%,180	10%, 231	6%, 135
Cache size 4096	9%, 220	5%, 120	3%, 68	2%, 43	1%, 28
Cache size 8192	8%, 193	5%, 120	3%, 68	2%, 43	1%, 28

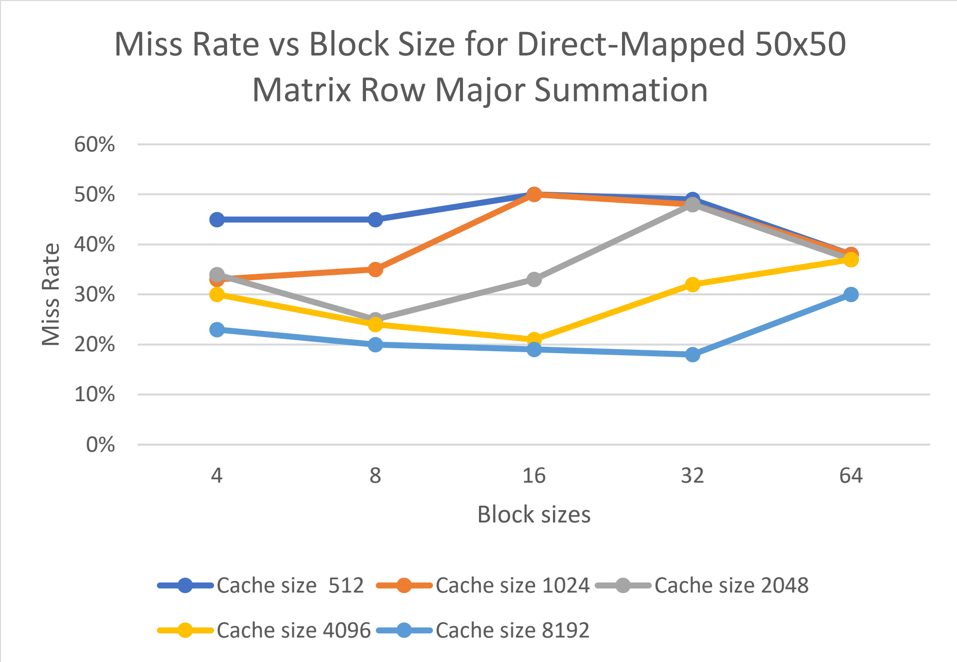
Table for Direct-Mapped 25x25 matrice, Column Major Summation



The first size I tried was a 25x25 matrix. The results seemed to align with what the book suggests on theoretical results, with an exception for values with lower cache sizes, specifically cache size 512 and 1024. Why did values with cache size had such greater miss rates and why does the pattern of it seem different than the others and the theoretical graph (in that it follows a similar pattern to $-x^2$ when the others and the theoretical graph have an x^2-adjacent pattern)? My assumption is that, since for cache size 512 to be preserved, the block sizes require a much fewer corresponding number of blocks (8 blocks for block size 16, 4 blocks for block size 32, and so on). This makes the rate of a possible miss much higher. As the cache size increases, the miss rate and the pattern of it starts to stabilize and starts to follow a closer pattern to each other, and the miss rate seems to lower as cache size increases in general. Another reason for the patterns being irregular could be due to the cache size of the MARS environment and/or how the computer I conducted this experiment on handles different caches. The computer I use has a 4GB RAM, so it should be enough theoretically, however I have no idea of how much cache space MARS has.
The miss counts for column-major summation are greater than the ones for row-major summation and the reason for that is that the program worked in this order: initialization of matrix -> row-summation -> column-summation -> display part of the code.
The miss rates for column-major summation tend to be lower than the ones for row-major summation and this is because the row-major summation goes through a complicated loop while column-major summation only keeps adding the addresses and the values on the respective addresses. Their rates are not quite as different as they are in the next experiment done with a size 50x50 matrix.

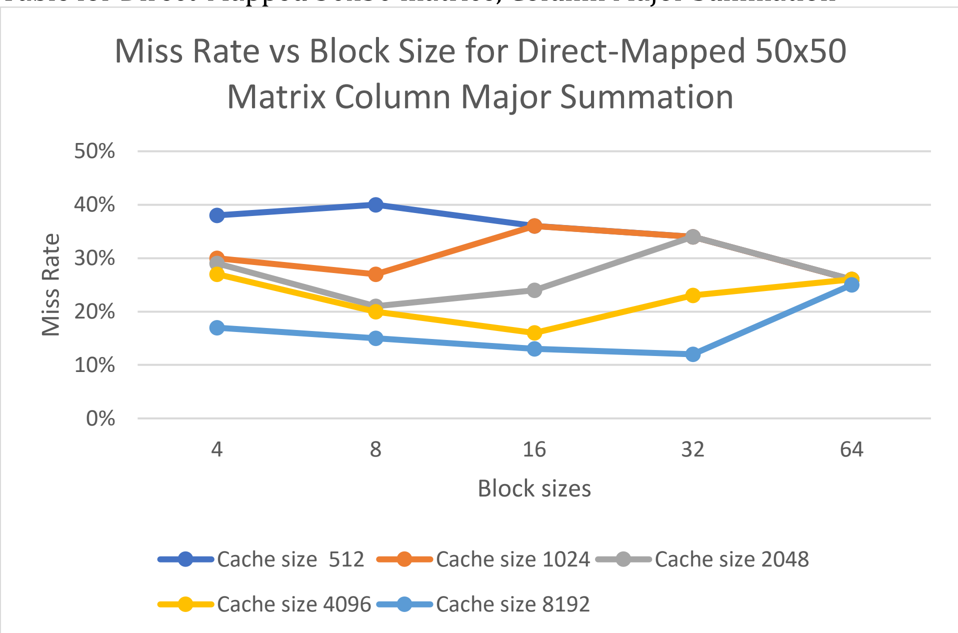
	Block size 4	**Block size 8**	**Block size 16**	**Block size 32**	**Block size 64**
Cache size 512	45%, 2440	45%, 2447	50%, 2697	49%, 2632	38%, 2069
Cache size 1024	33%, 1754	35%, 1867	50%, 2685	48%, 2608	38%, 2035
Cache size 2048	34%, 1728	25%, 1343	33%, 1791	48%, 2599	37%, 2018
Cache size 4096	30%, 1639	24%, 1297	21%, 1124	32%, 1722	37%, 2012
Cache size 8192	23%, 1245	20%, 1086	19%, 1004	18%, 962	30%, 1631

Table for Direct-Mapped 50x50 matrice, Row Major Summation



	Block size 4	Block size 8	Block size 16	Block size 32	Block size 64
Cache size 512	38%, 3075	40%, 3173	36%, 2869	34%, 2715	26%, 2111
Cache size 1024	30%, 2387	27%, 2187	36%, 2847	34%, 2691	26%, 2077
Cache size 2048	29%, 2342	21%, 1656	24%, 1952	34%, 2682	26%, 2060
Cache size 4096	27%, 2163	20%, 1566	16%, 1264	23%, 1798	26%, 2054
Cache size 8192	17%, 1385	15%, 1163	13%, 1050	12%, 992	25%, 1982

Table for Direct-Mapped 50x50 matrice, Column Major Summation



The second experiment was done on a 50x50 matrix. For all cache sizes, the miss rates seem to result in a higher percentage than it was on the same cache sizes on the 25x25 experiment. The results again seem to align with what the book suggests on theoretical results, with the exception of lower cache sizes. My assumption again is that lesser block sizes require a much fewer corresponding number of blocks and that it causes the rate of a possible miss to go higher. Again as the cache size increases, the miss rate and the pattern of it starts to stabilize and starts to follow a closer pattern to each other, and the miss rate seems to lower as cache size increases generally. The miss counts for column-major summation are greater than the ones for row-major summation for these tables too and the reason is as stated in the above explanation. Again, for some irregularities of patterns in graphs, MARS environment and/or the computer itself could possibly be to blame.

The miss rates for column-major summation are much lower than the ones for row-major summation and this is because the row-major summation goes through a complicated loop while column-major summation only keeps adding the addresses and the values on the respective addresses. The difference in the miss rates of row-major and column-major summation is greater than it is on the 25x25 matrix.

b)

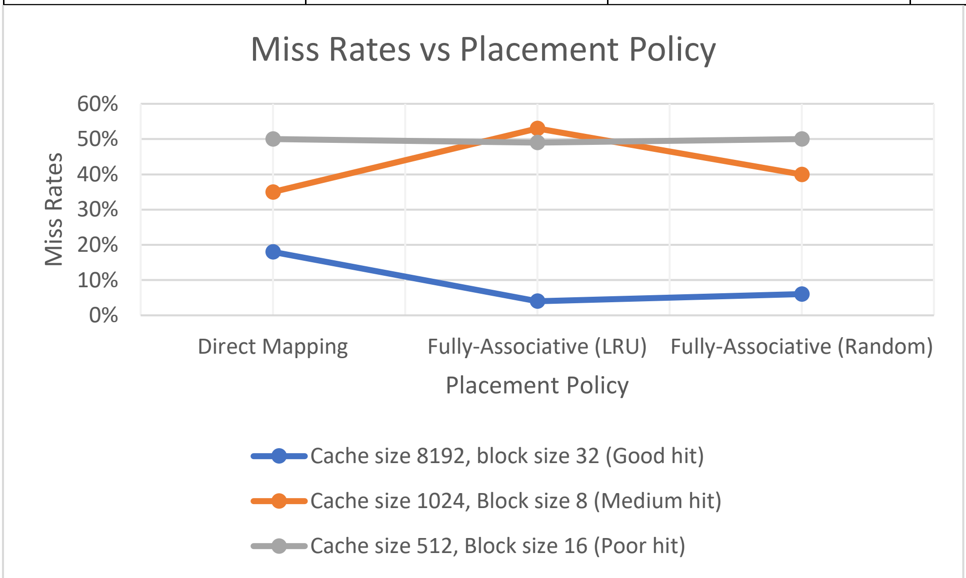
I choose all of the values from 50x50 matrix row-major summation table:

Good hit: Cache size 8192, Block size 32

Medium hit: Cache size 1024, Block size 8

Poor hit: Cache size 512, Block size 16

	Direct Mapping	Fully-Associative (LRU)	Fully-Associative (Random)
Cache size 8192, block size 32 (Good hit)	18%, 962	4%, 191	6%, 318
Cache size 1024, Block size 8 (Medium hit)	35%, 1867	53%, 2828	40%, 2161
Cache size 512, Block size 16 (Poor hit)	50%, 2697	49%, 2667	50%, 2681



Even though it seemed to make a positive difference for the good hit configuration, the result seemed to be much worse for hit rates in the medium configuration and didn't change much for the poor hit configuration, although it decreased the total miss counts in small amounts.

c)All the configurations are from the 50x50 matrix row-major summation table.

Cache size 1024, Block size 8 (Medium hit - 35% miss rate, 962 miss count on Direct Mapping)

N	N- way Set Associative
1	35%, 1867
2	39%, 2085
4	39%, 2114
8	40%, 2144

The set size increase seemed to result in a higher miss rate, much like how the Fully-Associative placement policy affected it. However, it is easy to see that N-way set associative placement policy did not make the miss rates go as worse as it did in the experiment for Fully-Associative placement policy which makes it a better substitute since it has actually made a change for positive in this case. Increase in set sizes make the miss rates and miss counts higher than they already were and set size = 1 seems to make the best case overall in terms of a higher hit rate, however if we need to compare inbetween the changed ones (since set size = 1 is actually the one we see on direct mapping), set size = 2 seems to result in a better hit rate than the other changes done.

Cache size 8192, block size 32 (Good hit - 18% miss rate, 962 miss count on Direct Mapping)

N	N- way Set Associative
1	18%, 962
2	9%, 502
4	6%, 297
8	5%, 280

The hit rates got better in this experiment, which was expected from the previous experiment seeing that it got a better hit rate with a more efficient placement policy. Miss rates seem to result in a similar way with the previous experiment, and the increase in set sizes seem to make the hit rate get better and the miss rate get lower and lower. Set size = 8 seems to make the best case for this example.

Cache size 512, Block size 16 (Poor hit - 50% miss rate, 2697 miss count on Direct Mapping)

N	N- way Set Associative
1	50%, 2699
2	50%, 2692
4	50%, 2690
8	50%, 2684

In a similar fashion to how the other two resulted, this experiment also resulted in a same way with the previous Fully-Associative placement policy experiment in which the miss rates did not seem to change. One change we can observe is that, although the overall rate does not change, the miss counts actually seem to get lower, which means that hit counts have gotten higher – this shows that N-way set associative placement policy was actually better for the hit rates in this case. Set size = 8 seemed to result in the best case even though all of the cases seem to result in the same miss rates.