23.12.2020

BILKENT UNIVERSITY

COMPUTER ENGINEERING DEPT.

CS224 LAB 6 SECTION 2

LAB REPORT

ZÜBEYİR BODUR

21702382

Fall 2020

1. Cache Memory Problems & Program

a) Question 1

Assuming that the main memory size is 0.5 GB, we will have 29 bits of memory address and the table asked in part 1 will be the following (answers are red colored).

No.	Cache Size (KB)	N-Way Cache	Word Size (bits)	Block size (# words)	# of Sets	Tag Size (bits)	Set Size (bits)	Block Offset Size (bits)	Byte Offset Size (bits)	Block Replacement Policy Needed (Yes/No)
1	4	1	32	4	**256**	**17**	**8**	**2**	**2**	**No**
2	4	2	32	4	**128**	**18**	**7**	**2**	**2**	**Yes**
3	4	4	32	8	**64**	**18**	**6**	**3**	**2**	**Yes**
4	4	Full	32	8	**1**	**24**	**0**	**3**	**2**	**Yes**
5	32	1	16	4	**1024**	**16**	**10**	**2**	**1**	**No**
6	32	2	16	4	**512**	**17**	**9**	**2**	**1**	**Yes**
7	32	4	8	16	**512**	**16**	**9**	**4**	**0**	**Yes**
8	32	Full	8	16	**1**	**25**	**0**	**4**	**0**	**Yes**

b) Question 2

C = 16 words, b = 4 words and N = 1 is given.

i. 2-a

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

ii. 2-b

C = 16 words, b = 4 words and N = 1.

We know that B = C / b, S = B / N. Then, we get C = N * S * b. We find number of sets as S = C / (N * b) = 16 words / (1 * 4 words) = 4.

In MIPS, 1 word = 32 bits = 4 bytes

Block offset size = $\log_2 b = \log_2 4 = 2$

Set offset size = $\log_2 S = \log_2 4 = 2$

Byte offset size = $\log_2$(single word size in bytes) = $\log_2 4 = 2$

Tag size = 32 - (byte offset size + set offset size + block offset size) = 32 - 6 = 26

A single set has 26 + 1 + 4 * (32) = 155 bits

The cache has 155 * 4 = 620 bits

iii. 2-c

1 EQUALITY COMPARATOR to check if tags are equal

1 AND gate for anding the valid bit and output of the comparator.

1 4:1 MULTIPLEXER to select the correct data according to byte offset.

c) Question 3

C = 8 words, b = 1 words and N = 2 is given and we know that block replacement policy is LRU. Assuming the MIPS code in question 2, the answers are below.

i. 3-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

ii. 3-b

C = 8 words, b = 1 words and N = 2.

We know that B = C / b, S = B / N. Then, we get C = N * S * b. We find number of sets as S = C / (N * b) = 8 words / (2 * 1 words) = 4.

In MIPS, 1 word = 32 bits = 4 bytes

Block offset size = $\log_2 b = \log_2 1 = 0$

Set offset size = $\log_2 S = \log_2 4 = 2$

Byte offset size = $\log_2$(single word size in bytes) = $\log_2 4 = 2$

Tag size = 32 - (byte offset size + set offset size + block offset size) = 32 - 4 = 28

A single set has (28 + 1 + 32) * 2 + 1 = 123 bits

The cache has 123 * 4 = 492 bits

iii. 3-c

2 EQUALITY COMPARATOR to check if tags are equal in Way0 & Way1.

2 AND gate for anding the valid bits and output of the comparators, to get Hit_1 and Hit_0.

1 OR gate for oring the Hit_1 and Hit_0.

1 2:1 MULTIPLEXER to select the correct data according to Hit_1.

d) Question 4

The MIPS program for this question is below:

```
# CS224 Lab6
# Part 1 Question 4
# Author : Zübeyir Bodur
# ID : 21702382
# MIPS Program for a square Matrix
	.text

ui_loop:
	la	$a0, endl
	jal	print_str
	# PRINT 1
	la	$a0, first_option
	jal	print_str
	# PRINT 2
	la	$a0, second_option
```

```
	jal		print_str
	# PRINT 3
	la		$a0, third_option
	jal		print_str
	# PRINT 4
	la		$a0, fourth_option
	jal		print_str
	# PRINT 5
	la		$a0, fifth_option
	jal		print_str
	# ASK INPUT
	jal		input_int
	# BRANCH INPUTS
	beq		$v0, 1, init
	beq		$v0, 2, display
	beq		$v0, 3, rowsum
	beq		$v0, 4, colsum
	beq		$v0, 5, endui_loop
	j		ui_loop
	init:
		la		$a0, ask_n
		jal		print_str
		jal		input_int
		move		$s1, $v0	# store N in s1
		move		$a0, $s1
		jal		createMatrix	# allocate storage from heap
		move		$s0, $v0	# store arr address in s0
		move		$a0, $s0
		move		$a1, $s1
		jal		initMatrix		# fill matrix
		j		ui_loop
	display:
		move		$a0, $s0
		move		$a1, $s1
		jal		displayMatrix
		j		ui_loop
	rowsum:
		move		$a0, $s0
		move		$a1, $s1
		jal		rowMajorSum
		move		$s2, $v0	# s2 will store the sum
		la		$a0, result
		jal		print_str
		move		$a0, $s2
		jal		print_int
		j		ui_loop
	colsum:
		move		$a0, $s0
		move		$a1, $s1
		jal		colMajorSum
		move		$s2, $v0	# s2 will store the sum
		la		$a0, result
		jal		print_str
		move		$a0, $s2
		jal		print_int
		j		ui_loop
endui_loop:
li		$v0, 10
syscall
```

```
#=====SUBPROGRAMS=======

# Create an array of size N^2
# to represent a square matrix NxN
# Arguments = >
#      $a0 - N, number of rows & columns
#      in the NxN square matrix
# Returns = >
#      $v0 - address of the array
createMatrix:
        mul     $a0, $a0, $a0       # n := n * n
        mul     $a0, $a0, 4         # n := 4 * n
        li      $v0, 9
        syscall
        jr      $ra

# Initialize the matrix in the given address
# the contents will be ....
# 1          2          3          ..     N
# N+1        N+2        N+3        ..     2N
# 2N+1       2N+2       2N+3       ..     3N
# .          .          .          ..     .
# .          .          .          ..     .
# (N-1)N+1   (N-1)N+2   (N-1)N+3   ..     N^2
# However, the array will start from (1, 1), follows the
# rows and ends at N^2.
# Arguments = >
#      $a0 - address of the array
#      $a1 - N
# Returns = >
initMatrix:
        mul     $a1, $a1, $a1           # N := N^2
        addi $t0, $0, 1
        addi $a1, $a1, 1
        blt     $t0, $a1, loop_0        # for ($t0 = 1; $t0 < N^2 + 1;...)
        j       endloop_0
loop_0:
        sw      $t0, 0($a0)             # arr[i] = $t0
        addi $a0, $a0, 4            # i++
        addi $t0, $t0, 1            # $t0++
        blt     $t0, $a1, loop_0
endloop_0:
        jr      $ra

# Computes the row major sum of the given
# NxN sqr. matrix
# in address $a0
# Arguments = >
#      $a0 - address of the array
#      $a1 - N
# Returns = >
#      $v0 - row major sum
rowMajorSum:
        addi $sp, $sp, -4
        sw      $ra, 0($sp)
        addi $v0, $0 , 0            # sum := 0

        addi $t0, $0, 1             # rowno := 1
```

```
	addi $a1, $a1, 1			# N := N + 1
	blt	$t0, $a1, loop_1		# for (rowno = 1; rowno < N + 1;...)
	j	endloop_1
	loop_1:
		addi $t2, $0, 0	# rowsum := 0
		addi $t1, $0, 1	# colno := 1
		blt	$t1, $a1, loop_2	# for (colno = 1; colno < N + 1;...)
		j	endloop_2
		loop_2:
			addi $sp, $sp, -24
			sw	$a0, 0($sp)
			sw	$a1, 4($sp)
			sw	$a2, 8($sp)
			sw	$a3, 12($sp)
			sw	$v0, 16($sp)
			sw	$ra, 20($sp)
			addi $a1, $a1, -1
			move	$a2, $t0
			move	$a3, $t1
			jal	getVal			# cell = get(row, col)
			add	$t2, $t2, $v0	# rowsum += cell
			lw	$a0, 0($sp)
			lw	$a1, 4($sp)
			lw	$a2, 8($sp)
			lw	$a3, 12($sp)
			lw	$v0, 16($sp)
			lw	$ra, 20($sp)
			addi $sp, $sp, 24
			addi $t1, $t1, 1	# colno++
			blt	$t1, $a1, loop_2
		endloop_2:

		add	$v0, $v0, $t2		# sum += rowsum
		addi $t0, $t0, 1		# rowno++
		blt	$t0, $a1, loop_1
	endloop_1:
	lw	$ra, 0($sp)
	addi $sp, $sp, 4
	jr	$ra

# Computes the column major sum of the given
# NxN sqr. matrix
# in address $a0
# Arguments = >
#	$a0 - address of the array
#	$a1 - N
# Returns = >
#	$v0 - row major sum
colMajorSum:
	addi $sp, $sp, -4
	sw	$ra, 0($sp)
	addi $v0, $0 , 0		# sum := 0

	addi $t0, $0, 1		# colno := 1
	addi $a1, $a1, 1		# N := N + 1
	blt	$t0, $a1, loop_3	# for (colno = 1; colno < N + 1;...)
	j	endloop_3
	loop_3:
		addi $t2, $0, 0		# colsum := 0
```

```
            addi $t1, $0, 1                 # rowno := 1
            blt       $t1, $a1, loop_4      # for (rowno = 1; rowno < N + 1;...)
            j         endloop_4
            loop_4:
                addi $sp, $sp, -24
                sw        $a0, 0($sp)
                sw        $a1, 4($sp)
                sw        $a2, 8($sp)
                sw        $a3, 12($sp)
                sw        $v0, 16($sp)
                sw        $ra, 20($sp)
                addi $a1, $a1, -1
                move      $a2, $t1
                move      $a3, $t0
                jal       getVal            # cell = get(row, col)
                add       $t2, $t2, $v0     # colsum += cell
                lw        $a0, 0($sp)
                lw        $a1, 4($sp)
                lw        $a2, 8($sp)
                lw        $a3, 12($sp)
                lw        $v0, 16($sp)
                lw        $ra, 20($sp)
                addi $sp, $sp, 24
                addi $t1, $t1, 1          # rowno++
                blt       $t1, $a1, loop_4
            endloop_4:

            add       $v0, $v0, $t2     # sum += colsum
            addi $t0, $t0, 1          # colno++
            blt       $t0, $a1, loop_3
        endloop_3:
        lw        $ra, 0($sp)
        addi $sp, $sp, 4
        jr        $ra

# Displays the given NxN square matrix
# Arguments = >
#     $a0 - address of the array
#     $a1 - N
# Returns = >
displayMatrix:
        addi $t0, $0, 0  # index := 0
        mul       $t3, $a1, $a1     # N := N^2
        blt       $t0, $t3, loop_5      # for ($t0 = 0; $t0 < N^2;...)
        j         endloop_5
        loop_5:
            lw        $t1, 0($a0)       # $t1 := arr[index]
            addi $sp, $sp, -8
            sw        $a0, 0($sp)
            sw        $ra, 4($sp)
            move      $a0, $t1
            jal       print_int         # print arr[i]
            div       $t0, $a1
            mfhi $t2            # $t2 := index % N
            addi $t4, $a1, -1       # $t4 := N - 1

            beq       $t2, $t4, if_0
                la        $a0, tab   # print wspc otherwise
                j         endif_0
```

```
        if_0:
            la      $a0, endl  # print \n if end of row
        endif_0:
        jal     print_str
        lw      $a0, 0($sp)
        lw      $ra, 4($sp)
        addi $sp, $sp, 8
        addi $a0, $a0, 4        # next adress
        addi $t0, $t0, 1        # index++
        blt     $t0, $t3, loop_5
    endloop_5:
    jr      $ra

# Gets the value in (row, col)
# in the given matrix
# Arguments = >
#    $a0 - address of the array
#    $a1 - N
#    $a2 - row no.
#    $a3 - col no.
# Returns = >
#    $v0 - value read from matrix array
getVal:
    # compute offset = N * (row no. - 1) + col no. - 1
    addi $a2, $a2, -1
    addi $a3, $a3, -1
    mul     $a2, $a1, $a2
    add     $a2, $a2, $a3   # $a2 = # of indexes to add
    mul     $a2, $a2, 4     # $a2 = offset
    add     $a0, $a0, $a2
    lw      $v0, 0($a0)     # get the value
    jr      $ra

# Arguments = >
# Returns = >
#    $v0 - int read
input_int:
    li      $v0, 5
    syscall
    jr      $ra

# Arguments = >
#    $a0 - str to print
# Returns = >
print_str:
    li      $v0, 4
    syscall
    jr      $ra

# Arguments = >
#    $a0 - int to print
# Returns = >
print_int:
    li      $v0, 1
    syscall
    jr      $ra

        .data
first_option:       .asciiz "1- Create a square matrix with N # of rows\n"
```

```
second_option:     .asciiz "2- Display the square matrix\n"
third_option:      .asciiz "3- Display sum of elements row-major\n"
fourth_option: .asciiz "4- Display sum of elements column-major\n"
fifth_option:      .asciiz "5- Quit\n"
ask_n:                 .asciiz "Enter N : "
result:            .asciiz "Result : "
hyphen:            .asciiz " - "
endl:                  .asciiz "\n"
wspc:                  .asciiz " "
tab:               .asciiz "\t"
```

2. Experiments with Data Cache Parameters

a) Results for Matrix 1

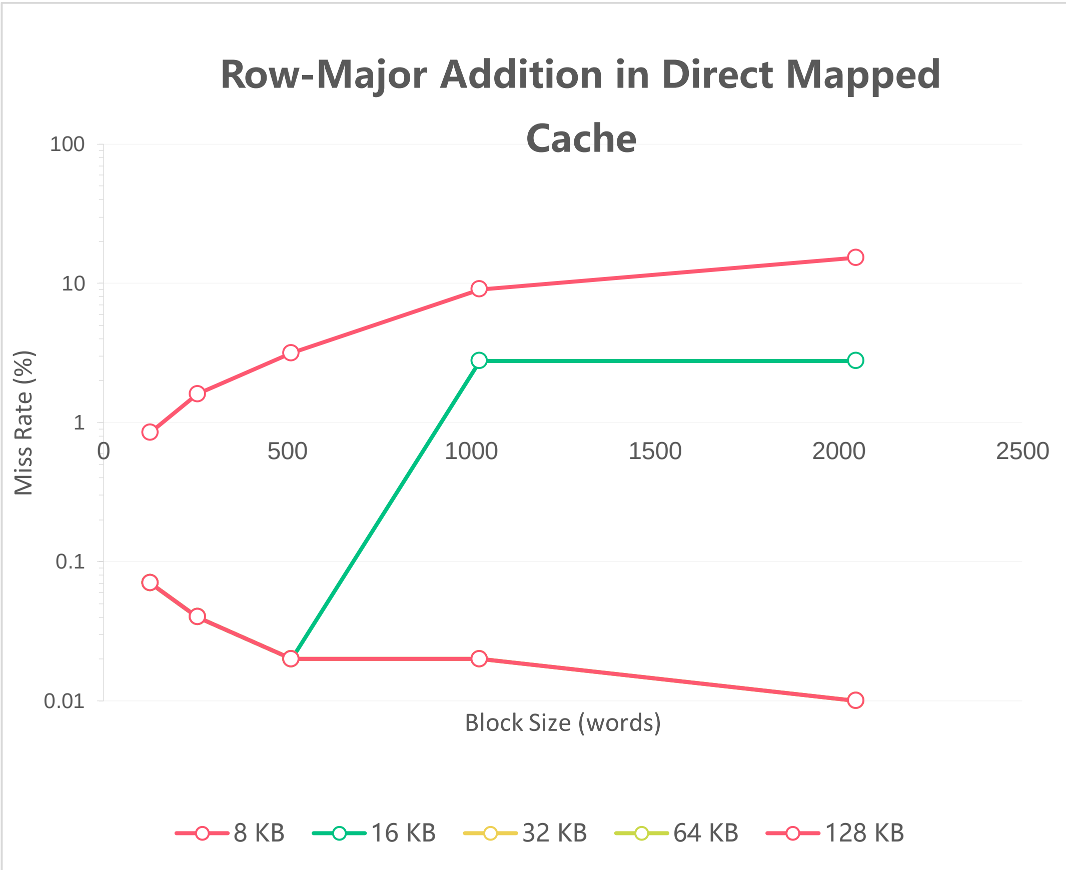
Below experiment is done with a 50x50 square matrix.

i. Direct Mapped Cache

The table below shows the miss rates and number of misses in a row major addition using direct mapped cache.

Cache Size (bytes)	Block size (words)				
	128	256	512	1024	2048
8 KB	277 Misses 0.85 % MR	523 Misses 1.60 % MR	1030 Misses 3.15 % MR	2955 Misses 9.04 % MR	5002 Misses 15.30 % MR
16 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	908 Misses 2.78 % MR	907 Misses 2.78 % MR
32 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR
64 KB	22 Misses 0.07 %	12 Misses 0.04 %	7 Misses 0.02 % MR	5 Misses 0.02 %	4 Misses 0.01 %

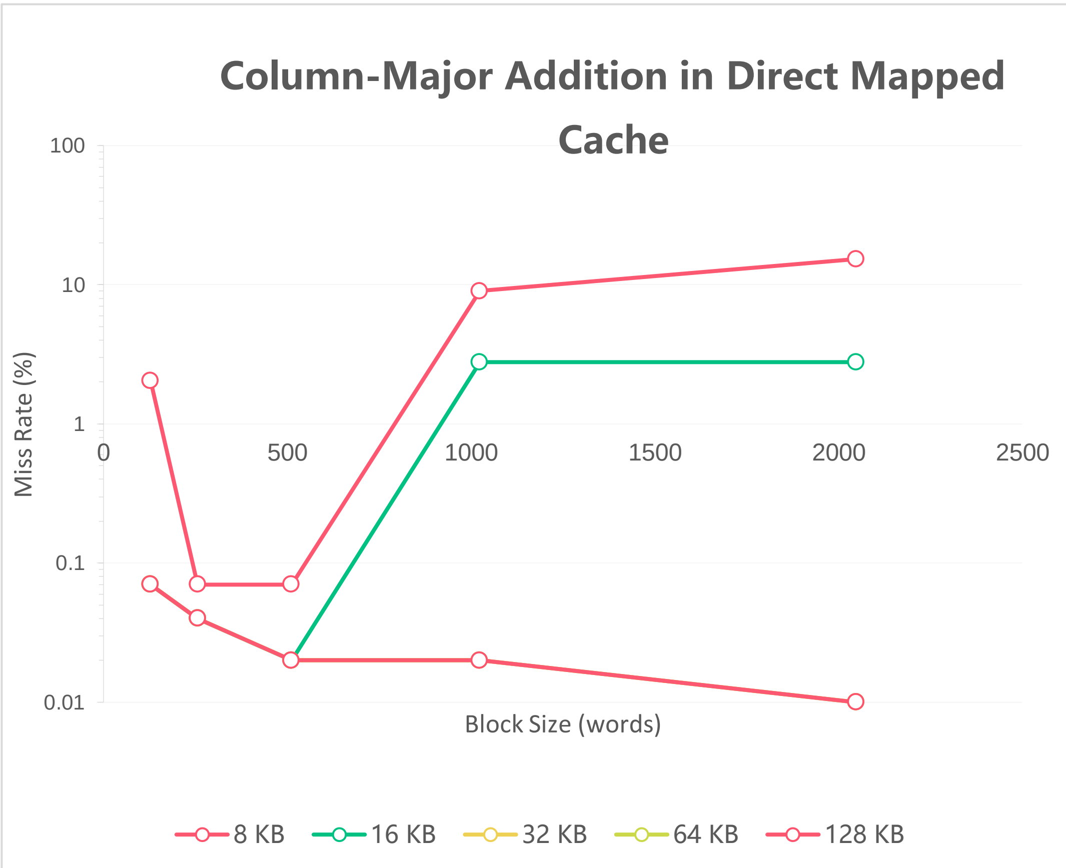
	MR	MR		MR	MR
128 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR



The table below shows the miss rates and number of misses in a column major addition using direct mapped cache.

Cache Size (bytes)	Block size (words)				
	128	256	512	1024	2048
8 KB	669 Misses 2.05 % MR	719 Misses 2.19 % MR	1128 Misses 3.45 % MR	2955 Misses 9.03 % MR	5002 Misses 15.30 % MR

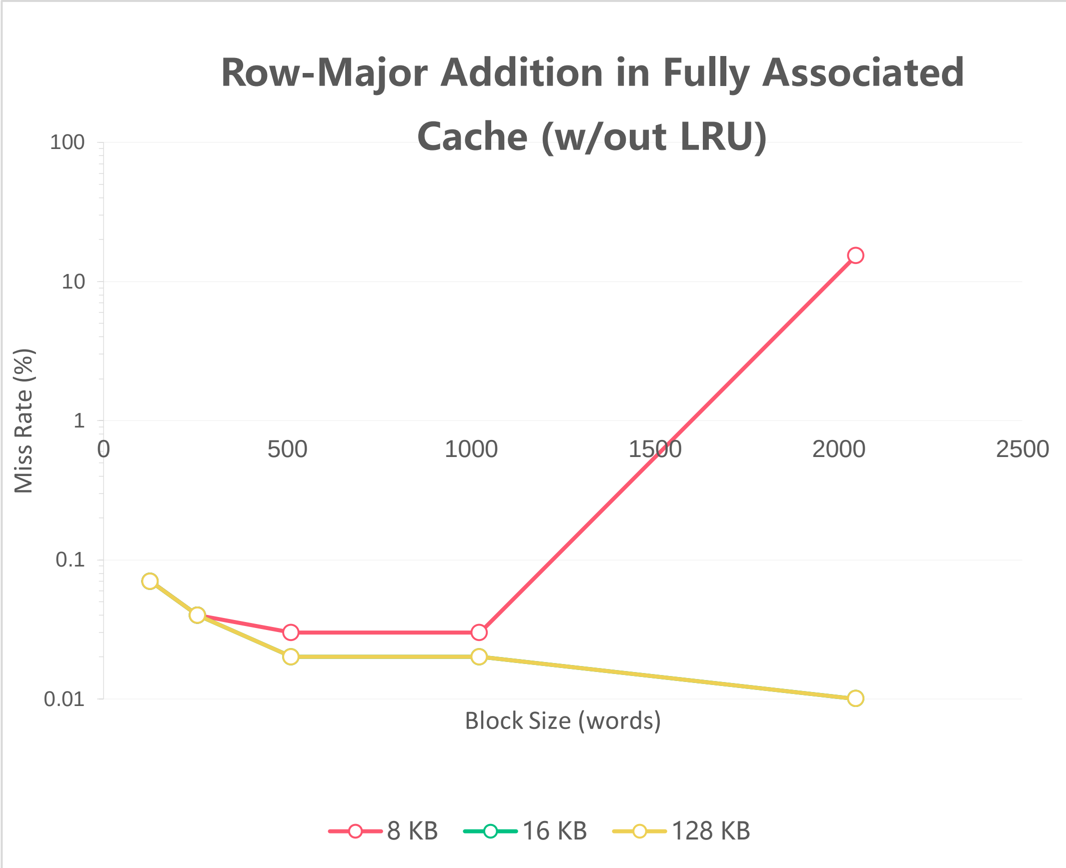
16 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	908 Misses 2.78 % MR	907 Misses 2.78 % MR
32 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR
64 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR
128 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR



ii. Fully Associative Cache, without LRU

The table below shows the miss rates and number of misses in a row major addition using fully associative cache without LRU block replacement policy - so it's totally random.

Cache Size (bytes)	Block size (words)				
	128	256	512	1024	2048
8 KB	23 Misses 0.07 % MR	12 Misses 0.04 % MR	10 Misses 0.03 % MR	10 Misses 0.03 % MR	5002 Misses 15.30 % MR
16 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR
128 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR

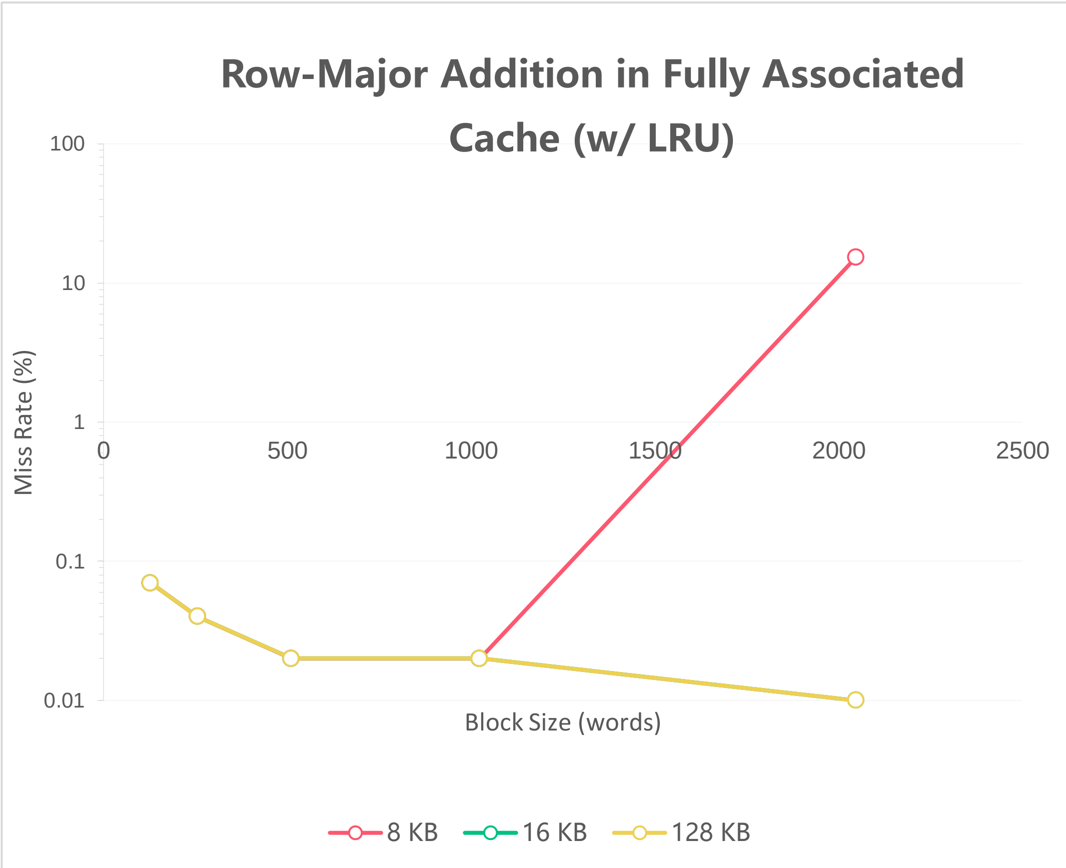


iii. Fully Associative Cache, with LRU

The table below shows the miss rates and number of misses in a row major addition using fully associative cache with LRU block replacement policy.

Cache Size (bytes)	Block size (words)				
	128	256	512	1024	2048
8 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	5002 Misses 15.30 % MR
16 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR

128 KB	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR



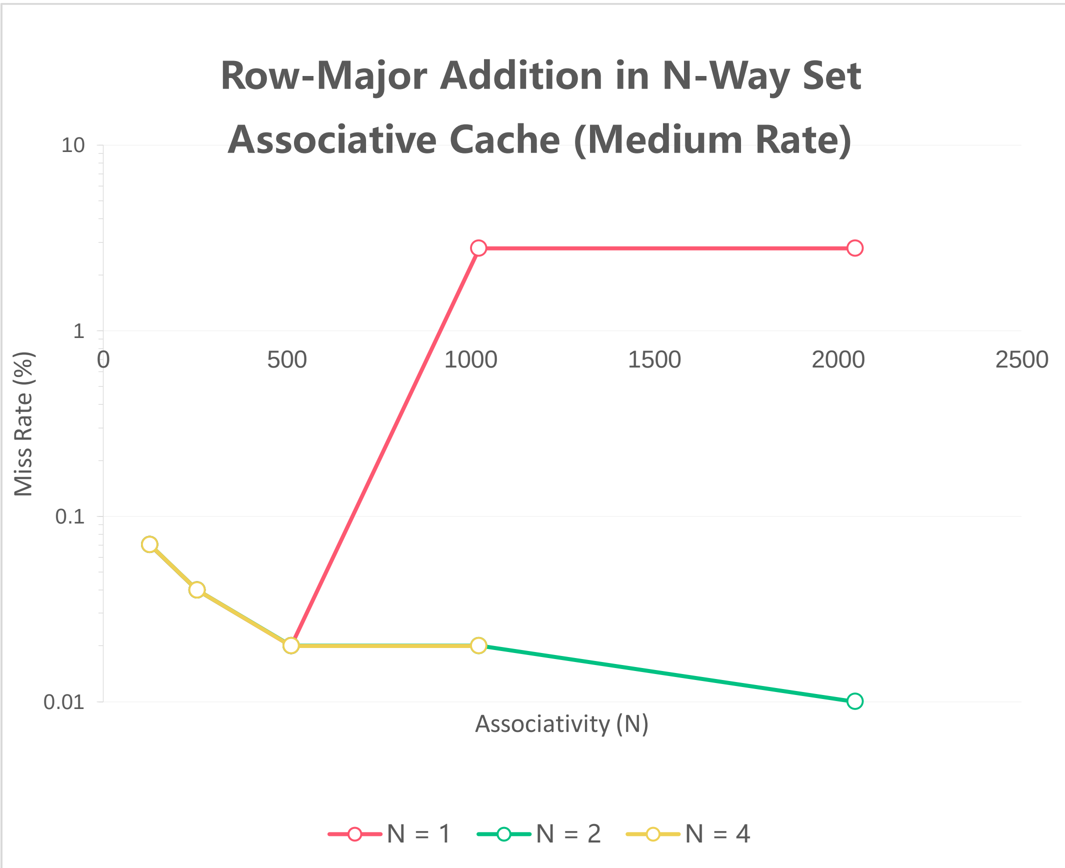
iv. N-Way Set Associative Cache

The table below shows the miss rates and number of misses in a row major addition in medium miss rate configuration for different set sizes; using n-way set associative cache with LRU block replacement policy.

Medium Miss Rate Config: 16 KB Cache size, 128-2048 block size

Set Size	Block size (words)				
	128	256	512	1024	2048

1	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	908 Misses 2.78 % MR	907 Misses 2.78 % MR
2	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	4 Misses 0.01 % MR
4	22 Misses 0.07 % MR	12 Misses 0.04 % MR	7 Misses 0.02 % MR	5 Misses 0.02 % MR	-



v. Observations

Direct mapped cache observations:

- As of implementation row major addition was no different than column major addition; only registers used for row and column numbers were replaced. However, column major summation had more miss rates than row major sum. The reason could be that the ordering of the accessing is changed in a way that it caused the data to be stored in the same sets.

- If the cache size is too small, miss rate increases as the block size increases (direct mapped cache graph - 8 KB).
- If the cache size is too large, miss rate decreases as the block size increases (all 128 KB caches).
- If it's in between, it first decreases, then it increases.

Fully associative cache observations:

- Fully associated architecture had effect only on small size (8 KB) cache (poor miss rate result). It decreased the number of misses for small block sizes, but for large block sizes (2048 words), there were capacity conflicts.
- Using LRU had almost no effect in fully associated caches. It decreased the # of misses in 8 KB cache for a very small amount.

Using n-way set associative cache had very good effect to prevent conflict misses. For more than 1 set sizes, there were a significant decrease in misses for 1024 and 2048 word block sizes. The results for more than 1 set size were almost the same, so we can say that there were no "best" associativity.

b) Results for Matrix 2

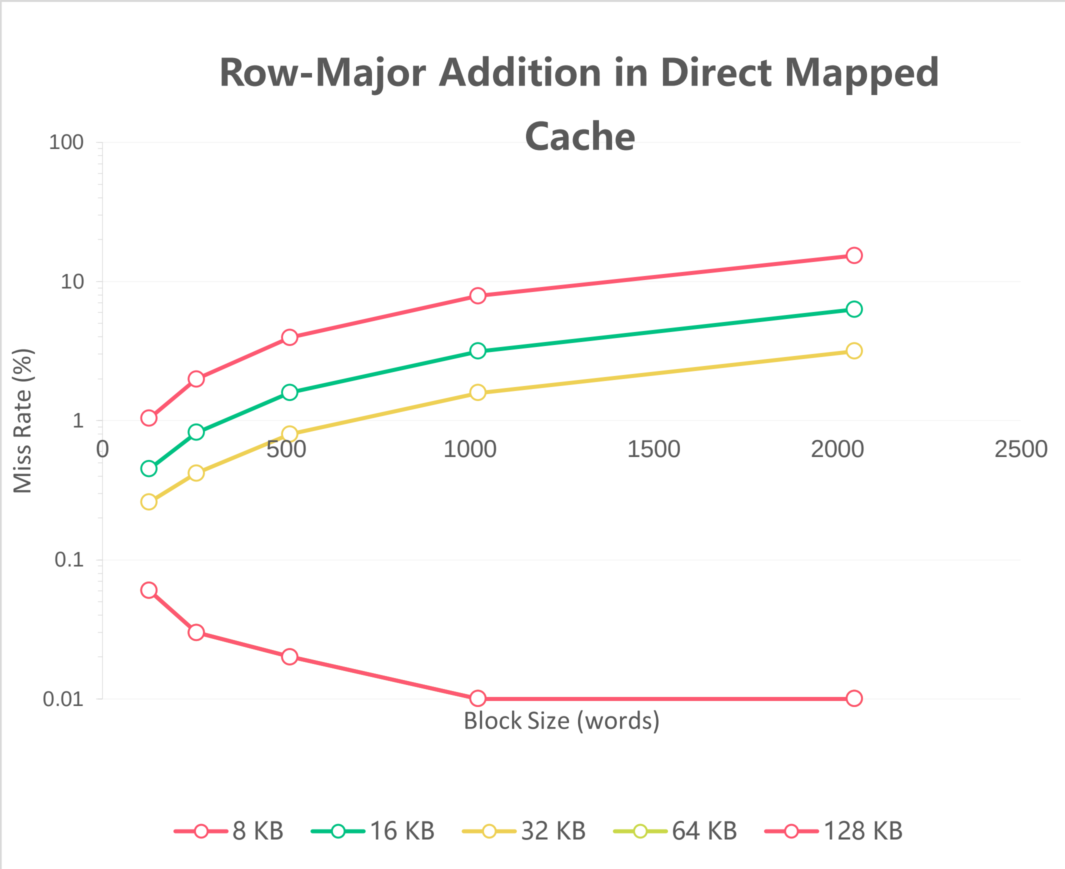
Below experiment is done with a 100x100 square matrix.

i. Direct Mapped Cache

The table below shows the miss rates and number of misses in a row major addition using direct mapped cache.

Cache Size (bytes)	**Block size (words)**				
	128	256	512	1024	2048

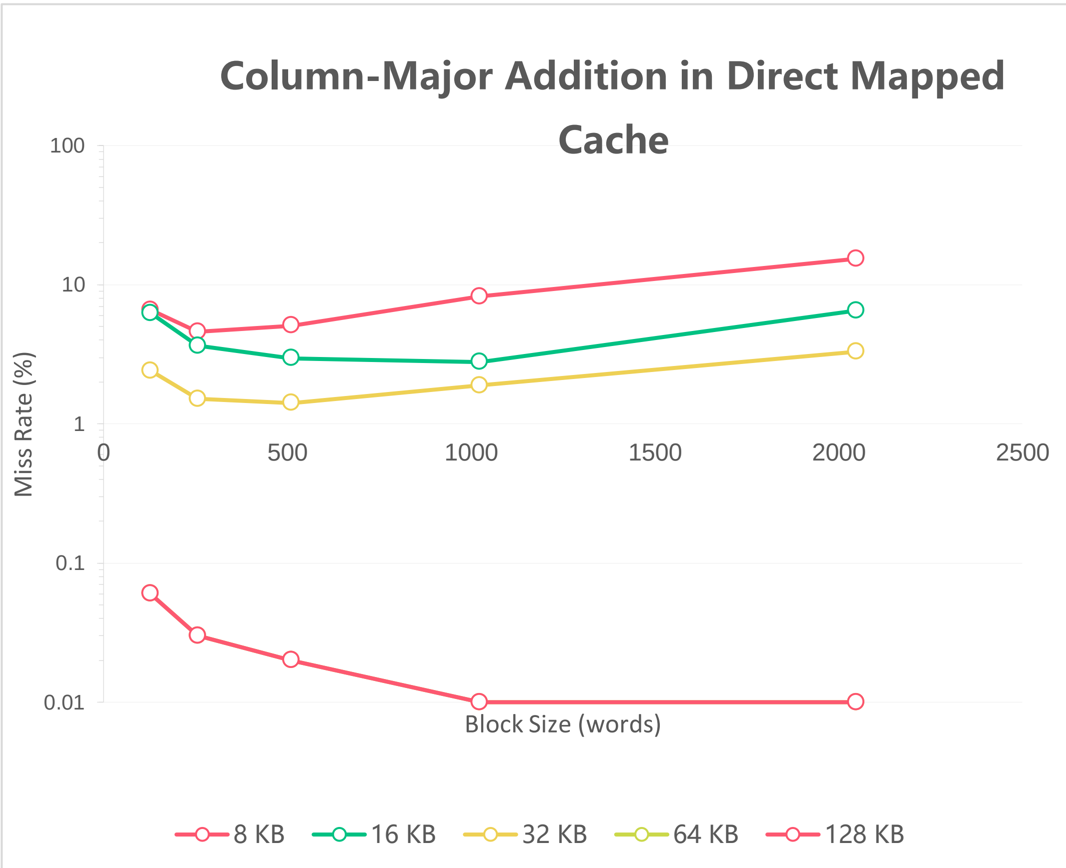
8 KB	1356 Misses 1.04 % MR	2597 Misses 1.99 % MR	5137 Misses 3.95 % MR	10247 Misses 7.87 % MR	20002 Misses 15.36 % MR
16 KB	591 Misses 0.45 % MR	1064 Misses 0.82 % MR	2068 Misses 1.59 % MR	4106 Misses 3.15 % MR	8197 Misses 6.30 % MR
32 KB	336 Misses 0.26 % MR	553 Misses 0.42 % MR	1045 Misses 0.80 % MR	2059 Misses 1.58 % MR	4102 Misses 3.15 % MR
64 KB	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	7 Misses 0.01 % MR
128 KB	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	7 Misses 0.01 % MR



The table below shows the miss rates and number of misses in a column major addition using direct mapped cache.

Cache Size (bytes)	**Block size (words)**				
	128	256	512	1024	2048
8 KB	8598 Misses 6.60 % MR	5978 Misses 4.59 % MR	6622 Misses 5.09 % MR	10742 Misses 8.25 % MR	20002 Misses 15.36 % MR
16 KB	8130 Misses 6.24 %	4742 Misses 3.64 %	3850 Misses 2.96 % MR	4898 Misses 2.78 %	8494 Misses 6.52 % MR

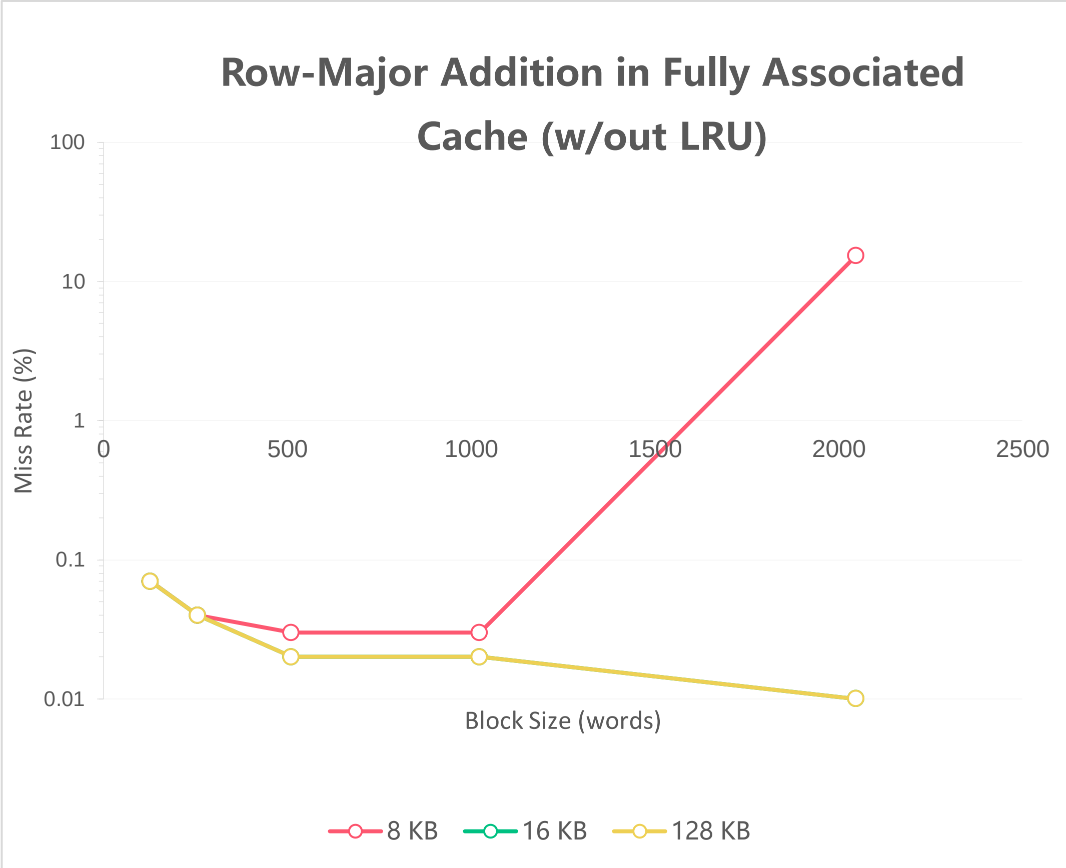
	MR	MR		MR	
32 KB	3138 Misses 2.41 % MR	1969 Misses 1.51 % MR	1837 Misses 1.41 % MR	2455 Misses 1.89 % MR	4300 Misses 3.30 % MR
64 KB	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	7 Misses 0.01 % MR
128 KB	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	7 Misses 0.01 % MR



ii. Fully Associative Cache, without LRU

The table below shows the miss rates and number of misses in a row major addition using fully associative cache without LRU block replacement policy - so it’s totally random.

Cache Size (bytes)	**Block size (words)**				
	128	256	512	1024	2048
8 KB	83 Misses 0.07 % MR	48 Misses 0.04 % MR	29 Misses 0.03 % MR	15 Misses 0.03 % MR	20002 Misses 15.30 % MR
16 KB	83 Misses 0.07 % MR	44 Misses 0.04 % MR	23 Misses 0.02 % MR	13 Misses 0.02 % MR	11 Misses 0.01 % MR
128 KB	81 Misses 0.07 % MR	42 Misses 0.04 % MR	22 Misses 0.02 % MR	12 Misses 0.02 % MR	7 Misses 0.01 % MR

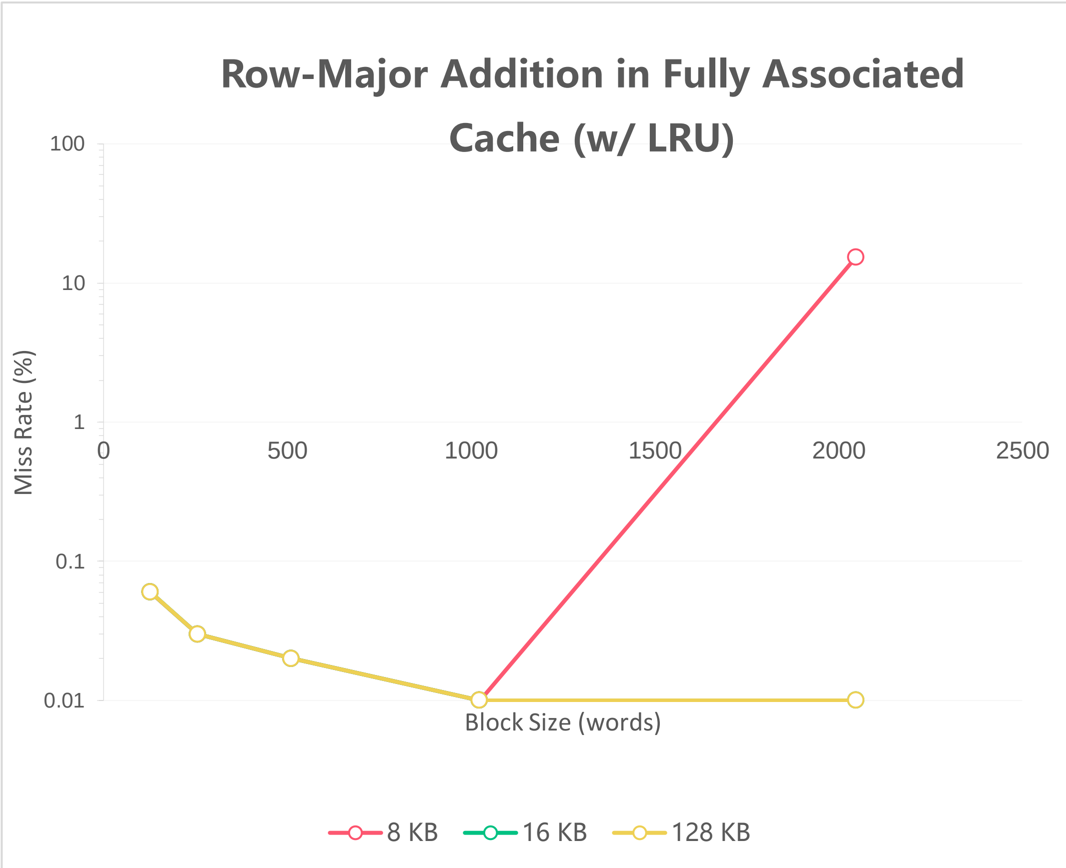


iii. Fully Associative Cache, with LRU

The table below shows the miss rates and number of misses in a row major addition using fully associative cache with LRU block replacement policy.

Cache Size (bytes)	**Block size (words)**				
	128	256	512	1024	2048
8 KB	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	20002 Misses 15.36 % MR
16 KB	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	7 Misses 0.01 % MR

128 KB	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	7 Misses 0.01 % MR

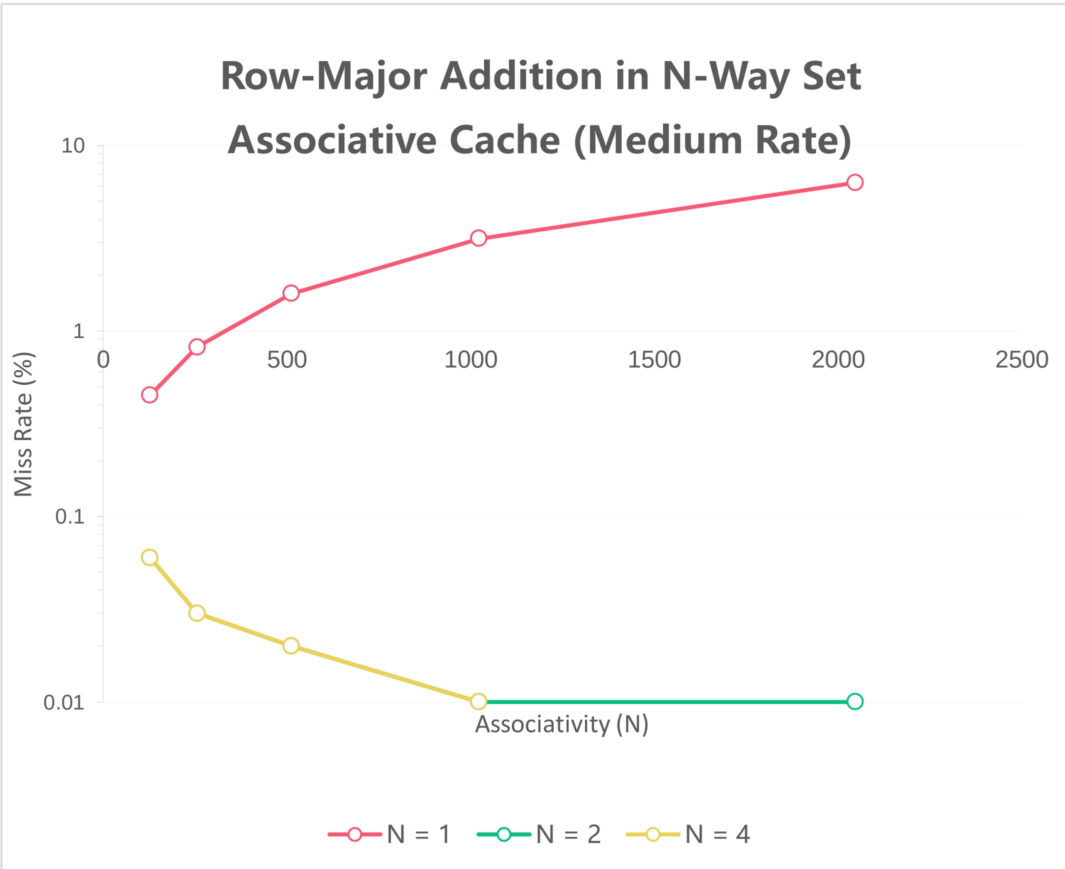


iv. N-Way Set Associative Cache

The table below shows the miss rates and number of misses in a row major addition in medium miss rate configuration for different set sizes; using n-way set associative cache with LRU block replacement policy.

Medium Miss Rate Configurations: 16 KB Cache size, 128-2048 block size

Set Size	Block size (words)				
	128	256	512	1024	2048
1	591 Misses 0.45 % MR	1064 Misses 0.82 % MR	2068 Misses 1.59 % MR	4106 Misses 3.15 % MR	8197 Misses 6.30 % MR
2	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	7 Misses 0.01 % MR
4	81 Misses 0.06 % MR	42 Misses 0.03 % MR	22 Misses 0.02 % MR	12 Misses 0.01 % MR	-



v. Observations

Direct mapped cache observations:

- For row-major summation, miss rates increased except for 128 KB. This shows that for too small caches, miss rates increase as the block size increases.
- For column-major summation, how the ordering of accessing made a negative effect, increasing overall miss rates. However, the pattern followed by caches in column-major graph, except 128 KB, shows us that chosen cache sizes was close to expected results in the book.
  - From these results in column-major summation, we can infer that, for a cache with decent settings, the miss rates first decrease then increase as the block size increases.
- Overall, in the experiments we get increase in miss rates, it means that cache is starting to get full and getting conflict misses. Where we get decrease in miss rates, it means that if the cache is large enough by increasing the block size, we can decrease compulsory misses.

Full associative cache observations:

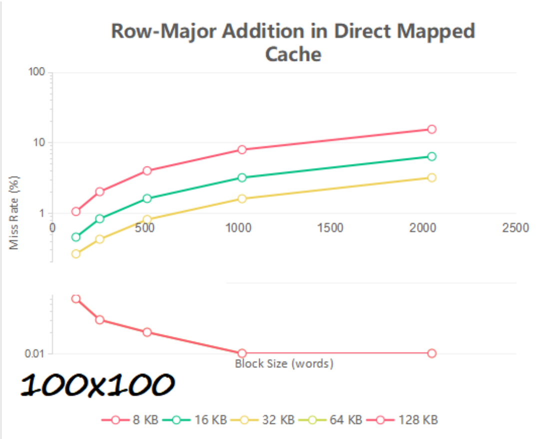
- Using LRU policy has no remarkable effect in fully associative caches. Because if the cache doesn't get capacity misses, it starts to get conflict misses since the number of sets are one.
- Using full associative cache instead of direct mapped cache gives better results.

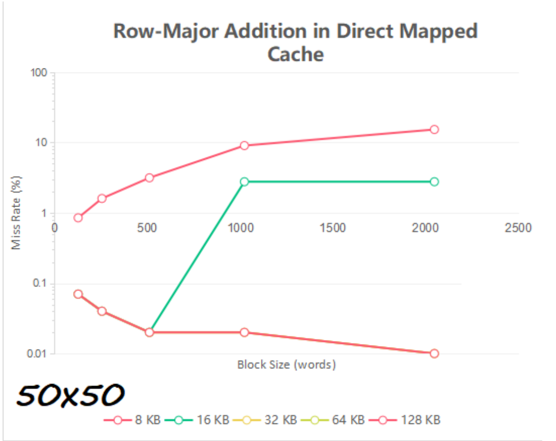
N-way set associative cache (w/ LRU) observations:

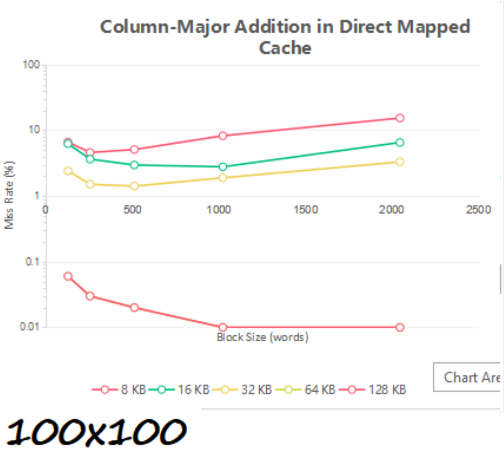
- If N = 1, it's simply a directed cache with LRU. LRU policy has a good effect and results in overall decrease in miss rates
- If N > 1, along with LRU, it makes a significant effect. If we use 8 KB cache and with N = 2, number of misses decrease by more than half. However, increasing N more than 2 doesn't give us better hit rates.

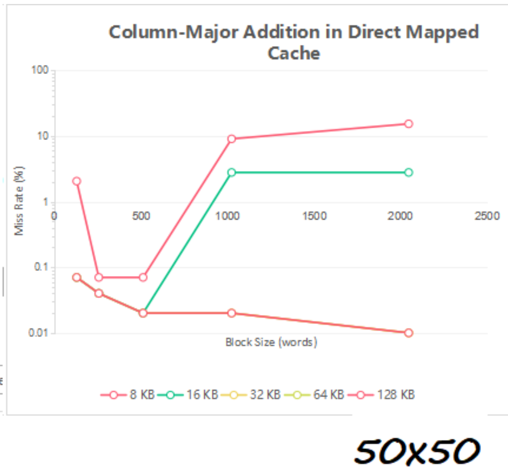
c) Conclusion

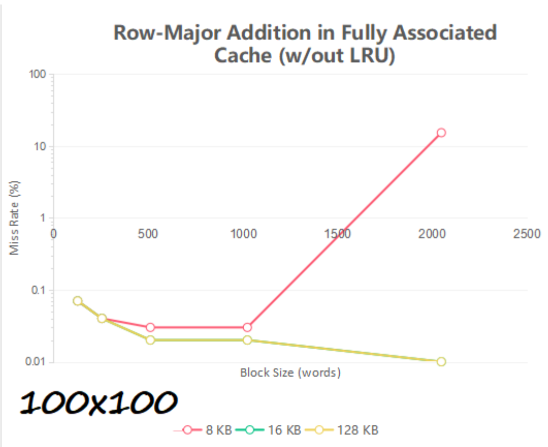
Before moving on to the conclusion, we can put the graphs for 50x50 and 100x100 and visually compare them.

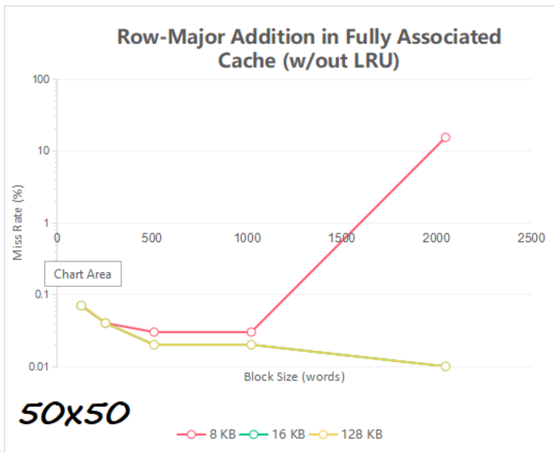


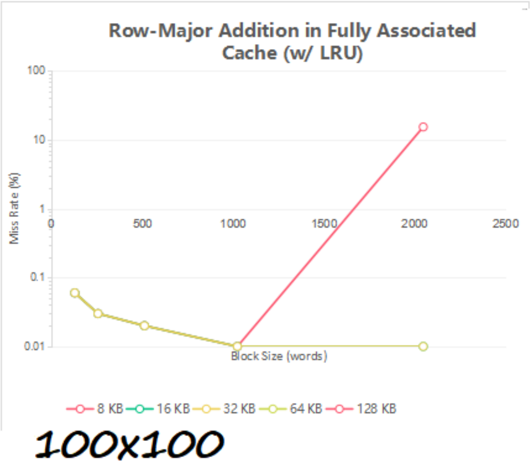


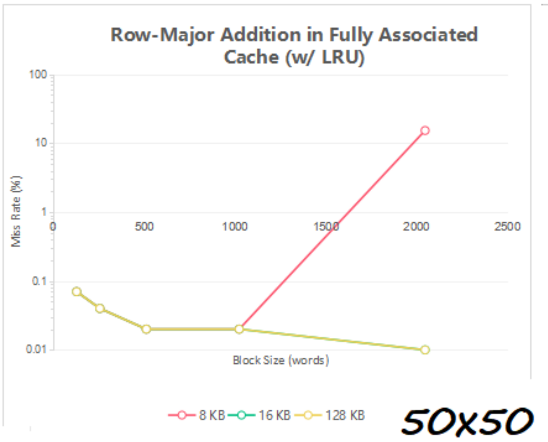


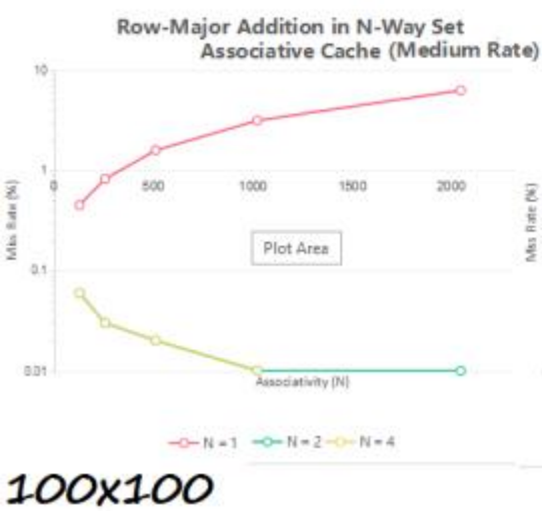


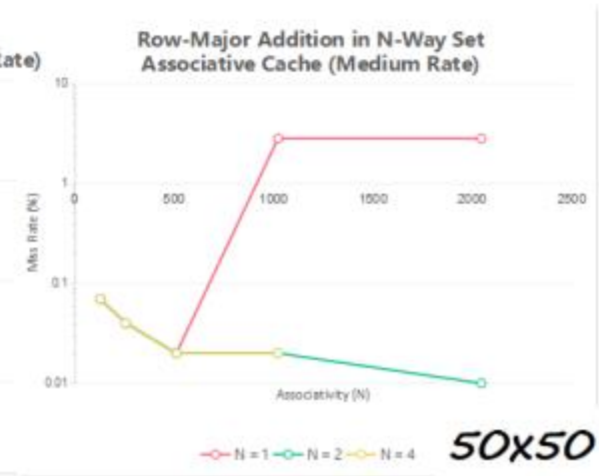












From this comparisons, we can directly observe that doing this experiment in 100x100 matrix is more accurate than 50x50 with selected cache and block sizes. We can infer this from the direct mapped cache comparisons. The reason is that, in 100x100 matrix, all caches except 128 KB follows the expected result given in the book. 50x50 matrix, however, only 16 KB cache in row-major addition and 8-16KB caches in column major addition follows this pattern. Hence, increasing the number of accesses gives more accurate results.

The second thing to observe between 100x100 vs 50x50 is that the cache's capacity gets more full in 100x100 matrix. From all the graphs, we can see that, in 50x50, there is a fall of before the rise, meaning the cache is starting to get full and it is having conflict misses. Conflict misses occur when the cache capacity is full, then we can infer that operations done in 100x100 matrix, obviously, consumes more data.