Sidharth Gilela
CMPE 110
11/20/15
Homework #3

1a) I have shown my tag and data arrays below to show as work:

	Tag Array	(Before)
Index	Valid	Tag
0		0x1
1		0x1
2		
...		

		Data Array	(Before)	
Index	Word1	Word2	Word3	Word4
0	(0x100)	(0x104)	(0x108)	(0x10c)
1	(0x110)	(0x114)	(0x118)	(0x11c)
2				
...				

After the tag for index 1 changes

	Tag Array	(After)
Index	Valid	Tag
0		0x1
1		0x2
2		
...		

		Data Array	(After)	
Index	Word1	Word2	Word3	Word4
0	(0x100)	(0x104)	(0x108)	(0x10c)
1	(0x210)	(0x214)	(0x218)	(0x21c)
2				
...				

This is the list of hits and misses for iteration1 and iteration2 which was calculated by the above information:

Address	Instruction	Iteration 1	Iteration 2
	loop:		
0x108	addiu r1, r1,-1	compulsory miss	hit
0x10c	addiu r2, r2, 1	hit	hit
0x110	j foo	compulsory miss	conflict miss
	...		
	foo:		
0x218	addiu r6, r6, 1	conflict miss	conflict miss
0x21c	bne r1, r0, loop	hit	hit

1b) 1st Iteration
Hit = 2
Compulsory miss = 2
Conflict miss = 1

2nd Iteration to 64th Iteration
Hit = 3
Conflict miss = 2

64(5) = Number of instructions = 320 instructions
2 + 63(3) = Number of hit instructions = 191 instructions
3 + 63(2) = Number of miss instructions = 129 instructions

Miss rate = (Number of misses/Total number of instructions)
= (129)/(320) = 0.403125 x 100 = 40.3125%

Average Memory Access Latency = Hit + Miss rate x Miss penalty

Hit = 1 cycle

Miss rate = 40.3125%

Miss penalty = 5 cycles

=> Average Memory Access Latency = 1 + 0.403125 x 5

= 3.015625 cycles/instruction

What kind of miss dominates?

Since only during the first iteration there were 2 compulsory misses and the during the rest of the iterations there were conflict misses, conflict misses have dominated the average memory access latency.

1c) If we were to predict how the cache performance would change if we replace the eight-entry, direct-mapped cache with an eight-entry, two-way, set-associative cache, we know one main thing. The main thing we know is that there will be more hits in this new type of cache since, each of the indexes can hold more than one type of set of addresses.

I used data tables for only set 0 and set 1 because these were the only sets used. I didn't need to use tag tables because I was able to understand the concept and come up with the answer with only the data tables. Here are the data tables below:

	Data Table	(Index 0)	
Word1	Word2	Word3	Word4
(0x100)	(0x104)	(0x108)	(0x10c)
(unused)	(unused)	(unused)	(unused)

	Data Table	(Index 1)	
Word1	Word2	Word3	Word4
(0x110)	(0x114)	(0x118)	(0x11c)
(0x210)	(0x214)	(0x218)	(0x21c)

This is the list of hits and misses for iteration1 and iteration2:

Address	Instruction	Iteration 1	Iteration 2
	loop:		
0x108	addiu r1, r1,-1	compulsory miss	hit
0x10c	addiu r2, r2, 1	hit	hit
0x110	j foo	compulsory miss	hit
	...		
	foo:		
0x218	addiu r6, r6, 1	compulsory miss	hit
0x21c	bne r1, r0, loop	hit	hit

64(5) = Number of instructions = 320 instructions

2 + 63(5) = Number of hit instructions = 317 instructions

3 + 63(0) = Number of miss instructions = 3 instructions

Miss rate = (Number of misses/Total number of instructions)
= (3)/(320) = 0.009375 x 100 = 0.9375%

Average Memory Access Latency = Hit + Miss rate x Miss penalty
Hit = 1 cycle
Miss rate = 0.9375%
Miss penalty = 5 cycles
=> Average Memory Access Latency = 1 + 0.009375 x 5 = 1.047 cycles/instruction

What type of misses?
The kind of misses that would be present with this type of cache architecture is compulsory miss.

Observations of this type of cache in comparison to the direct-mapped cache:
- The average memory access latency is smaller for a two-way set-associative cache.
- There were fewer conflict misses in this two-way set-associative cache
- The miss rate for this two-way set-associative cache was smaller

2) Information and work deduced before attempting any parts of the problem:

512-KB => 512(1024) = 524288 bytes = cache size

bytes x words x cache lines = cache size
-We are given that each word is 4 bytes so bytes = 4
-We are given that the number of words per cache line is 64 so words = 64

To calculate cache lines:
Cache lines = cache size/(bytes x words)
= 524288/(4 x 64) = 524288/256
= 2048 cache lines

A cache sketch to go by from the deduced information of byte, words, and cache lines is presented below:

		Cache	(Sketch)			
Index	Word 1	Word 2	Word 3	Word 4	...	Word 64
	1	2	3	4	...	64
0					...	
1					...	
2					...	
3					...	
...	...	...	...	...	...	
2047						

2a) To calculate cacheline offset:
Each word is 4 bytes => 2 bits are necessary for the byte offset since 2^2 = 4
Thus: Byte Offset = 2 bits
Word Offset: 64 words => 6 bits are necessary for word offset since 2^6 =64
Thus Word Offset: 6 bits
cacheline offset bits = 6 + 2 = 8 bits
To calculate cacheline index:
There are 2048 cache lines. 2 to the ? power equals 2048 is the index offset
? = log base 2 of 2048 which is 11 bits
cacheline index bits = 11 bits
To calculate tag:
We know there are 32 bits in the address.
Thus 32 – 11 – 8 = Number of bits for tag
tag bits = 13

Tag	Cacheline Index	Cacheline Offset
13 bits	11 bits	8 bits

2b) To calculate cacheline offset:
Each word is 4 bytes => 2 bits are necessary for the byte offset since 2^2 = 4
Thus: Byte Offset = 2 bits
Word Offset: 64 words => 6 bits are necessary for word offset since 2^6 =64
Thus Word Offset: 6 bits
cacheline offset bits = 6 + 2 = 8 bits
To calculate cacheline index:
In a fully associative cache, there is no need for any bits for the index
cacheline index bits = 0
To calculate tag:
We know there are 32 bits in the address.
Thus 32 – 8 = Number of bits for tag
tag bits = 24

Tag	Cacheline Index	Cacheline Offset
24 bits	0 bits	8 bits

2c) To calculate cacheline offset:
Each word is 4 bytes => 2 bits are necessary for the byte offset since 2^2 = 4
Thus: Byte Offset = 2 bits
Word Offset: 64 words => 6 bits are necessary for word offset since 2^6 =64
Thus Word Offset: 6 bits
cacheline offset bits = 6 + 2 = 8 bits
To calculate cacheline index:
There are 2048 cachelines and 16 sets.
Thus, each set has 2048/16 cachelines which equals 128 cachelines for each set
2 to the ? power equals 128 is the index offset
? = log base 2 of 128 which is 7 bits
cacheline index bits = 7 bits
To calculate tag:
We know there are 32 bits in the address.
Thus 32 – 8 – 7 = Number of bits for tag
tag bits = 17 bits

Tag	Cacheline Index	Cacheline Offset
17 bits	7 bits	8 bits

2d and 2e info)
The hexadecimal to binary conversion of each of the addresses is shown below:
0x128 = 0001 0010 1000
0xF40 = 1111 0100 0000
0xA00051 = 1010 0000 0000 0000 0101 0001
0x093 = 0000 1001 0011
0x4000B44 = 0100 0000 0000 0000 1011 0100 0100

2d) Using the binary digits of the address, I was able to figure the index and the tag:

The cacheline index = middle 11 bits		Tag(leftmost 13 bits)	
0x128 =>	0x001	0x128 =>	0x0000
0xF40 =>	0x00F	0xF40 =>	0x0000
0xA00051 =>	0x000	0xA00051 =>	0x0014
0x093 =>	0x000	0x093 =>	0x0000
0x4000B44 =>	0x00B	0x4000B44 =>	0x0080

From this information and the order of addresses from the table, I came up with a rough sketch of the data cache and tag array to help figure out the other parts to this question. Here are the sketches to my data cache and tag array:

		Cache	(Sketch)			
Index	Word 1	Word 2	Word 3	Word 4	...	Word 64
0						
1						
...						
B						
...						
F						
...						
Last line						

	Tag Array	(Sketch)	
Index	Valid	Tag	
0		0x0014	0x0000
1		0x0000	
...		...	
B		0x0080	
...		...	
F		0x0000	
...		...	
Last line			

Here is some information below that has helped me complete the table:

To figure out hit or miss?

⇨ used the data cache and the tag table

Modified?

⇨ when there is a write, there is modification

Caused replace?

⇨ Whenever a cacheline is changed such as a conflict miss

Data

⇨ There is a M for a read and a D for a write

Write-back to memory?

⇨ When the value in memory is changed. Caused when modified and replace happen at the same time.

With this information and the sketches above I was able to produce this table:

	Request	Cacheline					Caused	Write-back
Address	Type	Index	Hit or Miss?	Modified	Tag	Data	Replace?	to memory?
0x128	read	0x001	Compulsory miss	no	0x0000	M[0x100]	no	no
0xF40	write	0x00F	Compulsory miss	yes	0x0000	D[0xF00]	no	no
0xA00051	read	0x000	Compulsory miss	no	0x0014	M[0x000]	no	no
0x093	write	0x000	Conflict miss	yes	0x0000	D[0x000]	yes	yes
0x4000B44	read	0x00B	Compulsory miss	no	0x0080	M[0xB00]	no	no

2e) Using the binary digits of the address, I was able to figure the index and the tag:

The cacheline index = middle 7 bits

0x128 => 0x01
0xF40 => 0x0F
0xA00051 => 0x00
0x093 => 0x00
0x4000B44 => 0x0B

Tag(leftmost 17 bits)

0x128 => 0x00000
0xF40 => 0x00000
0xA00051 => 0x00140
0x093 => 0x00000
0x4000B44 => 0x00800

From the rough sketch of my data array and tag array in 2d, I was able to figure out the information in the table provided to the problem. The only difference about this 16-way set-associative cache is that there are now 16 sets of equally partitioned parts to the cache. These 16 parts make up the entire original cache. since all the indexes were small and the tags were figured out, I was able to come up with the information in the below table only using the information provided here:

To figure out hit or miss?
⇨ used the data cache and the tag table
Modified?
⇨ when there is a write, there is modification
Caused replace?
⇨ Whenever a cacheline is changed such as a conflict miss
Data
⇨ There is a M for a read and a D for a write
Write-back to memory?
=> When the value in memory is changed. Caused when modified and replace happen at the same time.

From all of this information, I was able to complete the table shown below:

	Request	Cacheline					Caused	Write-back
Address	Type	Index	Hit or Miss?	Modified	Tag	Data	Replace?	to memory?
0x128	read	0x01	Compulsory miss	no	0x00000	M[0x100]	no	no
0xF40	write	0x0F	Compulsory miss	yes	0x00000	D[0xF00]	no	no
0xA00051	read	0x00	Compulsory miss	no	0x00140	M[0x000]	no	no
0x093	write	0x00	Compulsory miss	yes	0x00000	D[0x000]	no	no
0x4000B44	read	0x0B	Compulsory miss	no	0x00800	M[0xB00]	no	no

2f) In order to calculate the overhead and actual size of a direct-mapped cache the work is shown below:
Cacheline size = 64 words x 4 bytes x 8 bits = 2048 bits
Afterwards, => tag + valid + modified = 13 + 1 + 1 = 15 bits
Overhead => 2048 = 2K x 15 = 30K overhead => 3.75KB
Actual = 512KB + 3.75KB = 515.75 KB

In order to calculate the overhead and actual size of a 16-way set-associative cache the work is shown below:
Cacheline size = 64 words x 4 bytes x 8 bits = 2048 bits
Afterwards => tag + valid + modified + 4 bits for indexing through sets
= 17 + 1 + 1 + 4 = 23 bits
Overhead = 2K x 23 = 46K overhead => 5.75KB
Actual = 512KB + 5.75KB = 517.75KB

Yes, the structure changes the overhead in terms of the number of memory bits because as shown above, the overhead for the 16-way set-associative cache is different than that of the direct-mapped cache.

3a) In order to figure out which machine has a faster execution time and the speedup I made an assumption that the x86 has 100 instructions and the MIPS has 125 instructions. However, with any number of instructions, the speedup will be the same. Also the computer that has a faster execution time will always have the faster execution time no matter the number of instructions.

Computer A with MIPS

Execution Time = CPI x Number of instructions x cycle time

Number of instructions = 125

CPI = Base CPI which is 1

Cycle time = 1/2GHz = 0.5 ns

Execution Time = 1cycle/instruction x 125instructions x (1/2GHz)

= 62.5 ns

Computer B with x86

Execution Time = CPI x Number of instructions x cycle time

Number of instructions = 100

CPI = Base CPI which is 2

Cycle time = 1/3GHz

Execution Time = 2cycle/instruction x 100instructions x (1/3GHz)

= 200/3 ns

Which machine has a faster execution time?

The execution time for computer A will always be faster than the execution time of computer B as proven above, for any number of instructions.

Speedup

(200/3)/(62.5) = 1.0667 => 1.0667 – 1 = 0.0667 => 0.0667 x 100 = 6.67%

The speedup of computer A over computer B is about 6.67% for any number of instructions.

3b) Computer A

Execution Time = Number of instructions x CPI x cycle time

Number of instructions = 125

Cycle time = 1/2GHz = 0.5 ns

CPI

CPI = Base CPI + Instruction CPI + Data CPI

Base CPI = 1

Instruction CPI = hit time(L1) + miss rate(L1) x miss penalty(L1)

Hit time(L1) = instruction hit time = 1

Miss penalty(L1) = Main memory access time = 250 cycles

Miss rate(L1) = 2% = 0.02

Info = 70% not loads and stores

=> Instruction CPI = 0.7(1+0.02(250)) = 4.2

Data CPI = hit time(L1) + miss rate(L1) x miss penalty(L1)

Info = 30% are loads/stores

=> Data CPI = 0.3(1+(0.08)(250)) = 6.3

CPI = 1 + 6.3 + 4.2 = 11.5 cycles/instruction

Execution Time = 125instructions x 11.5cycles/instruction x (1/2GHz)

= 718.75 ns

Computer B

Execution Time = Number of instructions x CPI x cycle time

Number of instructions = 100

Cycle time = 1/3GHz

CPI

CPI = Base CPI + Instruction CPI + Data CPI

Base CPI = 2

Instruction CPI = hit time(L1) + miss rate(L1) x miss penalty(L1)

Hit time(L1) = instruction hit time = 1

Miss penalty(L1) = Main memory access time = 300 cycles

Miss rate(L1) = 2% = 0.02

Info = 70% not loads and stores

=> Instruction CPI = 0.7(1+0.02(300)) = 4.9

Data CPI = hit time(L1) + miss rate(L1) x miss penalty(L1)

Info = 30% are loads/stores

=> Data CPI = 0.3(2+(0.05)(250)) = 5.1

CPI = 1 + 4.9 + 5.1 = 12 cycles/instruction

Execution Time = 100instructions x 12cycles/instruction x (1/3GHz)

= 400 ns

Which machine has a faster execution time?

The execution time for computer B will always be faster than the execution time of computer A as proven above, for any number of instructions.

Speedup

(718.75)/(400) = 1.7969 => 1.7969 – 1 = 0.7969 => 0.7969 x 100 = 79.69%

The speedup of computer B over computer A is about 79.69% for any number of instructions.

3c) Computer A

Execution Time = Number of instructions x CPI x cycle time

Number of instructions = 125

Cycle time = 1/2GHz = 0.5 ns

CPI

CPI = Base CPI + Instruction CPI + Data CPI

Base CPI = 1

Instruction CPI = hit time(L1) + miss rate(L1) x miss penalty(L1)

Hit time(L1) = instruction hit time = 1

Miss rate(L1) = 2% = 0.02

Miss penalty(L1) = hit time(L2) + miss rate(L2) x miss penalty(L2)

Hit time(L2) = 15 cycles

Miss rate(L2) = 3% = 0.03
Miss penalty(L2) = main memory access time = 250 cycles
Miss penalty(L1) = 15 + 0.03(250) = 22.5 cycles
Info = 70% not loads and stores
=> Instruction CPI = 0.7(1+0.02(22.5)) = 1.015 cycles
Data = hit time(L1) + miss rate(L1) x miss penalty(L1)
Info = 30% are loads/stores
=> Data CPI = 0.3(1+0.08(15+0.03(250))) = 0.84 cycles
CPI = 1 + 1.015 + 0.84 = 2.855 cycles/instruction
Execution Time = 125instructions x 2.855cycles/instruction x (1/2GHz)
= 178.4375 ns

Computer B
Execution Time = Number of instructions x CPI x cycle time
Number of instructions = 100
Cycle time = 1/3GHz
CPI
CPI = Base CPI + Instruction CPI + Data CPI
Base CPI = 2
Instruction CPI = hit time(L1) + miss rate(L1) x miss penalty(L1)
Hit time(L1) = instruction hit time = 1
Miss rate(L1) = 2% = 0.02
Miss penalty(L1) = hit time(L2) + miss rate(L2) x miss penalty(L2)
Hit time(L2) = 12 cycles
Miss rate(L2) = 4% = 0.04
Miss penalty(L2) = main memory access time = 300 cycles
Miss penalty(L1) = 12 + 0.04(300) = 24 cycles
Info = 70% not loads and stores
=> Instruction CPI = 0.7(1+0.02(24)) = 1.036 cycles
Data = hit time(L1) + miss rate(L1) x miss penalty(L1)
Info = 30% are loads/stores
=> Data CPI = 0.3(2+0.05(12+0.04(300))) = 0.96 cycles
CPI = 2 + 1.036 + 0.96 = 3.996 cycles/instruction
Execution Time = 100instructions x 3.996cycles/instruction x (1/3GHz)
= 133.2 ns

Which machine has a faster execution time?
The execution time for computer B will always be faster than the execution time of computer A as proven above, for any number of instructions.

Speedup
(178.4375)/(133.2) = 1.3396=> 1.3396 – 1 = 0.3396 => 0.3396 x 100 = 33.96%
The speedup of computer B over computer A is about 33.96% for any number of instructions.

4a and 4b) For the table information for both parts of this questions, I have calculated the individual parts such as CPI, etc in each of the question parts of 4a and 4b.

4a) The pipeline diagram I have made is shown below. Here is the pipeline diagram for two iterations:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
loop:																																
lw r12, 0(r4)	F	D	X	M	M	M	M	M	W																							
lw r13, 0(r5)		F	D	X	X	X	X	X	M	M	M	M	M	W																		
sw r12, 0(r5)			F	D	D	D	D	D	X	X	X	X	X	M	W																	
sw r13, 0(r4)				F	F	F	F	F	D	D	D	D	D	X	M	W																
addiu r14, r5, 0									F	F	F	F	F	D	X	M	W															
addiu r4, r4, 4														F	D	X	M	W														
addiu r5, r5, -4															F	D	X	M	W													
bne r4, r14, loop																F	D	X	M	W												
jr r31																	F	D	-	-	-											
nop																		F	-	-	-	-										
loop:																																
lw r12, 0(r4)																			F	D	X	M	W									
lw r13, 0(r5)																				F	D	X	M	W								
sw r12, 0(r5)																					F	D	X	M	W							
sw r13, 0(r4)																						F	D	X	M	W						
addiu r14, r5, 0																							F	D	X	M	W					
addiu r4, r4, 4																								F	D	X	M	W				
addiu r5, r5, -4																									F	D	X	M	W			
bne r4, r14, loop																										F	D	X	M	W		
jr r31																											F	D	-	-	-	
nop																												F	-	-	-	-

Extra information figured out:
- I was able to figure out that since an array each array spot is 4 bytes and the cacheline is 16 bytes, every first iteration of the first four iterations there will be a miss. Thus, out of the 32 iterations, there will be 8 cache misses.

This is how I calculated the information for the table:

Number of cycles = cycles for 1st iteration + cycles for 2nd to 32nd iteration + 2 cycles for last two nops
= 22(1) + 18(7) + 3(8)(10) + 2 = 390 cycles

Number of instructions = 8(32) = 256 instructions

CPI = cycles/instruction = 390 cycles/256 instructions = 1.523 cycles/instruction

Control Squashes = 2 squashes per iteration => 2(32) + last 2 nops = 66 cycles
= 66 cycles/256 instructions = 0.2578125 cycles/instruction

RAW Stalls = 0 cycles

Memory Stalls = 8 memory stalls every 4th iteration => 8(8) = 64 cycles
= 64 cycles/256 instructions = 0.25 cycles/instruction

Useful Work: 1.523 – 0.2578125 – 0.25 = 1.015 cycles/instruction

4b) The pipeline diagram I have made is shown below. Here is the pipeline diagram for two iterations:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
loop:																																									
lw r12, 0(r4)	F	D	X	M	M	M	M	M	W																															
lw r13, 0(r5)		F	D	X	X	X	X	M	M	M	M	M	W																											
sw r12, 0(r5)			F	D	D	D	D	X	X	X	X	X	M	W																										
sw r13, 0(r4)				F	F	F	F	D	D	D	D	D	X	M	W																									
addiu r14, r5, 0								F	F	F	F	F	D	X	M	W																								
addiu r4, r4, 4													F	D	X	M	W																						
addiu r5, r5, -4														F	D	X	M	W																					
bne r4, r14, loop															F	D	X	M	W																				
jr r31																F	D	X	-	-	-																		
nop																	F	D	-	-	-	-																	
loop:																																							
lw r12, 0(r4)																			F	D	X	M	M	M	M	M	W												
lw r13, 0(r5)																				F	D	X	X	X	X	X	M	M	M	M	M	W							
sw r12, 0(r5)																					F	D	D	D	D	D	X	X	X	X	X	M	W						
sw r13, 0(r4)																						F	F	F	F	F	D	D	D	D	D	X	M	W					
addiu r14, r5, 0																											F	F	F	F	F	D	X	M	W				
addiu r4, r4, 4																																F	D	X	M	W			
addiu r5, r5, -4																																	F	D	X	M	W		
bne r4, r14, loop																																		F	D	X	M	W		
jr r31																																			F	D	-	-	-	
nop																																				F	-	-	-	-

Extra information figured out:
- I was able to figure out that since a cacheline in a linked list can only contain 16 bytes and each node has 4 bytes with its forward pointer which is 4 bytes and previous pointer which is 4 bytes, each cacheline will have a miss. This means that during each iteration there will be a miss.

This is how I calculated the information for the table:
Number of cycles = cycles for 1st iteration + cycles for 2nd to 64th + 2 cycles for last 2 nops
= 22(1) + 18(31) + 2 = 582 cycles
Number of instructions = 8(32) = 256 instructions
CPI = cycles/instruction = 582 cycles/256 instructions = 2.273 cycles/instruction
Control Squashes = 2 squashes per iteration + last 2 nops cycles
= 2(32) + 2 = 66 cycles
= 66 cycles/256 instructions = 0.258 cycles/instruction
RAW Stalls = 0 cycles
Memory Stalls = 8 memory stalls every 4th iteration => 8(32) = 256 cycles
= 256 cycles/256 instructions = 1 cycles/instruction
Useful Work: 2.273 – 0.258 – 1 = 1.015 cycles/instruction

4c) The array data structure performs better in this specific example because it performs spatial locality and saves the overall amount of cycles. The execution time of the array would decrease if there were larger cachelines. This is because since the array performs with spatial locality, more bytes could fit into the cacheline resulting in more iterations being possible without a cache miss. However, if the cacheline size were to increase for linked lists, it wouldn't matter since, as said above, only one linked-list node gets allocated per cacheline. This wouldn't matter, because then, every iteration would yield a miss and thus, our current data structure with its specs does the exact same thing. The performance

would not change for a linked list. If we allowed multiple linked list nodes onto the same cacheline, then our performance would improve. This is because, as described with the array example above, we won't get a cache miss every iteration. This then will result in a lower amount of cycles, hence improving the performance of the cache for the linked list. The execution time would not change if the number of elements in the data structure grows asymptotically large. This is because the execution time is dependent on three things: the instruction count, the CPI, and the cycle time. The cycle time would stay the same and as the CPI increases, so does the number of instructions. This increase in the number of instructions in comparison to CPI is linear, otherwise known as O(n). Thus, since both these segments increase at the same rate, the execution time will be larger but when comparing it to the execution time of a smaller number of instructions, you will the growth of O(n). The dependency on the number of iterations causes a linear growth. If both data structures were already present in the cache such that there were no cache misses, the execution time would be better. There will be a a lower execution time because cycles were saved from a potential miss for both the array data structure and the linked list data structure. However, in the long run after the first iteration, for the linked list, there will be a miss at all the next iterations, since the cacheline size is still the same which causes problems. For the array, it will be the fifth iteration for the first miss, and then every 4^{th} iteration will be a miss afterwards. So, even though, the CPI will be lowered by a little, it will not cause a significant change to the execution time for both data structures. The data structures that make extensive use of dynamic memory allocation and pointers will have a different cache behavior than regular array or matrix-based data-structures. These data structures that use dynamic memory allocation and pointers, will be able to implement better spatial locality and thus will be able to reduce the number of cache misses. This, in return, will create a better execution time and hence, are the results of using these types of data structures.