4 Array vs. List Cache Behavior

				CPI Breakdown			
Part	**Number of Instructions**	**CPI**	**Execution Time (cyc)**	**Useful Work**	**Raw Stalls**	**Control Squashes**	**Memory Stalls**
Part 4.A	256	1.5	384	1	0.25	0	0.25
Part 4.B	256	2.25	576	1	1	0	0.25

Figure 14: Execution Time for Reverse Operation on Array and Linked List Data Structures

For the cycle type:
u = cycle of useful work
r = cycle lost due to RAW stal
m = cycle lost due to memory stall
c = cycle lost due to control squashes

4.a Analyzing Performance of an Array Data Structure

Table on next page.
The loop runs 32 times. Therefore, the total number of instructions executed is $32 \times 8 = 256$ instructions. The first iteration (between the first two bold lines) shows the pipeline flow when the cache misses both loads. Because the cacheline is 16 bytes, these misses will occur every 4 loops. The number of cycles in this cache-miss loop is 18. For when both load words find a hit in the cache (between the latter two bold lines), the cycle count for the loop is 10. This occurs 3 times out of every 4 loops. Therefore, the total cycles for 4 loops is $18 + 3 \times 10 = 48$. This is then looped 8 times for a total of 32 loops. Therefore, the total number of cycles for the program (ignoring the first 4 instructions for setup) is $8 \times 48 = 384$. The CPI for the entire program is therefore $384/256 = 1.5$. Here is a breakdown of the CPI for each cycle type:

```
First Iteration Cycle Type Breakdown:
u = 8 cycles
m = 8 cycles
r = 0 cycle
c = 2 cycles
Second Iteration Cycle Type Breakdown:
u = 8 cycles
m = 0 cycles
r = 0 cycle
c = 2 cycles
Overall CPI Breakdown:
u = 8*8 + 24*8 = 256 cycles, 256/256 = 1.00 CPI
m = 8*8 + 24*0 =  64 cycles,  64/256 = 0.25 CPI
r = 8*0 + 24*0 =   0 cycles,   0/256 = 0.00 CPI
c = 8*2 + 24*2 =  64 cycles,  64/256 = 0.25 CPI
total                                = 1.50 CPI
```

Cycle type:					m	m	m	m	u	m	m	m	m	u	u	u	u	u	u	u	c	c	u	u	u	u	u	u	u	u	c	c	u
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
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													
opA																	F	D	-	-	-												
opB																		F	-	-	-	-											
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			
opA																											F	D	-	-	-		
opB																												F	-	-	-	-	
lw r12, 0(r4)																													F	D	X	M	W

Figure 15: Array Data Structure Pipeline Diagram

4.b Analyzing Performance of a Linked-List Data Structure

Table on next page.
The loop runs 32 times. Therefore, the total number of instructions executed is $32 \times 8 = 256$ instructions. The first iteration (between the first two bold lines) shows the pipeline flow when the cache misses both of the first two loads. The number of cycles in this cache-miss loop is 18. On the second iteration (between the latter two bold lines), those first 2 load words will still be misses because they are in a different 16-byte location due to each node taking up an entire cacheline. r4 and r5 are changed to point to new nodes on the previous iteration, so these new nodes will need to be brought into the cache. Therefore, the second iteration and all other iterations have the same cycle count as the first iteration. The total number of cycles for the program (ignoring the first 4 instructions for setup) is $32 \times 18 = 576$. The CPI for the entire program is therefore $576/256 = 2.25$. Here is a breakdown of the CPI for each cycle type:

```
All Iterations Cycle Type Breakdown:
u = 8 cycles
m = 8 cycles
r = 0 cycle
c = 2 cycles
Overall CPI Breakdown:
u = 32*8 = 256 cycles, 256/256 = 1.00 CPI
m = 32*8 = 256 cycles, 256/256 = 1.00 CPI
r = 32*0 =   0 cycles,   0/256 = 0.00 CPI
c = 32*2 =  64 cycles,  64/256 = 0.25 CPI
total                          = 2.25 CPI
```

Cycle type:					m	m	m	m	u	m	m	m	m	u	u	u	u	u	u	u	c	c	m	m	m	m	u	m	m	m	m	u	u	u	u	u	u	u	c	c	u
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	41
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																								
lw r4, 4(r4)														F	D	X	M	W																							
lw r5, 8(r5)															F	D	X	M	W																						
bne r4, r14, loop																F	D	X	M	W																					
opA																	F	D	-	-	-																				
opB																		F	-	-	-	-																			
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						
lw r4, 4(r4)																																F	D	X	M	W					
lw r5, 8(r5)																																	F	D	X	M	W				
bne r4, r14, loop																																		F	D	X	M	W			
opA																																			F	D	-	-	-		
opB																																				F	-	-	-	-	
lw r12, 0(r4)																																					F	D	X	M	W

Figure 16: Linked List Data Structure Pipeline Diagram

4.c Comparison of Data Structures

In this example, the regular array data structure performed better due to its better spacial locality. Its values are packed right next to each other, so each cacheline can hold 4 values. The linked list can only pack 1 value (node) in each cacheline, so on every loop, the cache needs to read from the memory to bring in the new nodes.
The execution time of the regular array will improve slightly with larger cachelines (in relation to 1/x where x is the cacheline size) and the CPI will approach 1.25. However, the linked list will not improve at all because we are still only allocating one node per cacheline. This means that on every iteration, we still need to fetch from the memory.
If we now organize multiple linked-list nodes on the same cacheline, the linked-list performance will increase significantly and the CPI will also approach 1.25. This is because we will not have to fetch from the memory on every iteration. With more nodes in each cacheline, the spatial locality increases, and therefore the cache performance also increases.
Assuming the cacheline size stays the same and that the cache size is infinite, the execution time will grow linearly with the number of elements in the data structure. As we had shown above, the steady state behavior stays the same for all iterations after the first iteration. This directly relates to the asymptotic behavior of O(n) for these two data structures.
If we had no cache misses, both execution times will improve and the CPI will be 1.25. This is because the memory accesses will all only take a single cycle and there will be no cycles lost due to memory stall, which was a significant part of the CPI.
The general conclusion is that spatial locality can significantly affect the program performance. Simple and compact data structures such as arrays or matrices can achieve better performance due to its ability to pack more values into a single cacheline. Dynamic and irregular structures such as list, trees, and graphs are much worse due to the possible sparsity of the data. This means that while traversing these structures, the next node may not be near the previous node, which will make it likely to cause a cache miss. These structures are also much more complex, so we are not able to pack as many of them into a cacheline as we can with the simple data structures. This further negatively impacts the performance of programs.