ECE 4750 PSET 3

Tim Yao (ty252)

Nov 6, 2015

1 Tree Network Topologies

Part	Topology	B_c	b (bits/cycle)	γ_{max}	Θ_{term} (bits/cycle)
Part 1.A	Baseline Tree Topology	2	32	2	16
Part 1.B	Fat-Tree Topology	8	32	1	32

Figure 1: Ideal Terminal Throughput for Tree Topologies

Part	Topology	H_D	H_r	t_r (cycles)	H_c	t_c (cycle)	L/b (cycle)	t_0 (cycle)
Part 1.A	Baseline Tree Topology	4	3	2	2	1	3	11
Part 1.B	Fat-Tree Topology	4	3	11/3	2	1	3	16

Figure 2: Zero-Load Latency for Tree Topologies

1.a Performance of Baseline Tree Topology

To calculate the minimum bisection channel count, I first found the minimum bisection cut. This happens to be the cut on the channels that connect the two level-2 routers. This cut cuts 2 channels, so the B_c is 2.

To calculate the max channel, there are 3 possibilities. It can be the channels connecting between the nodes and the level-1 routers, level-1 and level-2 routers, and between the two level-2 routers. Because we are dealing with uniform random routing, all channels of one level has the same load.

The channel load for channels between nodes and level-1 routers are 1. In uniform random routing, the amount traffic sent from and to a node is always equal to 1.

The channel load for channels between level-1 and level-2 routers are 1.5. To calculate this, I first look at the amount of traffic that is sent from one node to all other nodes that require crossing the this channel. For example, all traffic that go from node 0 to 2,3,4,5,6,7 need to cross this channel. Each unit of traffic is 1/8, so for each node that resides under (in the sense of a tree) that channel is 6/8. Because there are 2 nodes (ie. node 0 and 1) under that channel, the max channel load is 2*6/8 = 12/8 = 3/2.

The channel load for channels between level-2 routers are 2. We use the same idea from the previous calculation to calculate this max load. Each node on the left side needs to send to 4 nodes on the right side, therefore, each node contributes 4*1/8 = 1/2 unit of traffic. There are 4 nodes, so the max load is 4*1/2 = 2. From this, we see that the max channel load is between the two level 2 routers. And using the ideal terminal throughput equation, $\Theta_{term}=B_c/\gamma_{max}$, I calculated that $\Theta_{term}=32/2=16$.

The network diameter is 4 router hops. One example is the path going from node 0 to node 7. This minimum path takes 4 router hops.

The average number of router hops is calculated using the following method. The average number of router hops for going from 0 to itself and all other nodes is the same from 1 to itself and all other nodes, from 2 to itself and all other nodes, and etc. Therefore, we can calculate the overall average number of router hops by simply analyzing the traffic going from 1 node to itself and all other nodes. If we pick 0, then the analysis is as following:

0->0: 1
0->1: 1

0->2: 3
0->3: 3
0->4: 4
0->5: 4
0->6: 4
0->7: 4

The average is then: $(1+1+3+3+4+4+4+4)/8=24/8=3$ hops.

Because all of the routers are radix-2, the average per-hop router latency is 2.

The method to calculate the average channel hop count is the same as calculating the average router hop count.

0->0: 0
0->1: 0
0->2: 2
0->3: 2
0->4: 3
0->5: 3
0->6: 3
0->7: 3

The average is then: $(2+2+3+3+3+3)/8=16/8=2$ hops.

The average per-hop channel latency is 1 cycle as stated in the assumption.

Because the message is 96 bits and the channel width is 32 bits, the serialization latency is $96/32=3$ cycles.

Now, we can calculate the zero-load latency.

$t_0=H_r t_r + H_c t_c + L/b$

$t_0=3*2+2*1+3=11$ cycles

1.b   Performance of Fat-Tree Topology

We use the same approach as in the previous section to calculate the max channel load and ideal terminal throughput for the baseline tree topology. Again, the channels connecting nodes to level-1 routers have a load of 1. The total load on all channels between level-1 and level-2 routers are the same (because this is a uniforma random traffic pattern), but now there are twice the number of channels. Therefore, the channel load is $6/8=3/4$. The total load on all channels between the two level-2 routers are also the same, but because there are now 4 times the number of channels, the channel load is $1/2$. Therefore, the max channel load is on the channels between nodes and level-1 routers. The max channel load is 1 and the ideal terminal throughput is $32/1=32$.

Because the layout topology and channel width of the fat-tree is the same as the baseline, the network diameter, average router hop count, average channel hop count, average per-hop channel latency, and serialization latency are the same.

The only difference here is the average per-hop router latency due to different radix routers. To calculate this, we take a similar approach to calculating the average hop counts. We use a weighted average for each node-pair due to the different latencies of different routers.

0->0: 3
0->1: 3
0->2: $(3+5+3)/3 = 11/3$
0->3: $(3+5+3)/3 = 11/3$
0->4: $(3+5+5+3)/4 = 16/4 = 4$
0->5: $(3+5+5+3)/4 = 16/4 = 4$
0->6: $(3+5+5+3)/4 = 16/4 = 4$
0->7: $(3+5+5+3)/4 = 16/4 = 4$

The average is: $(3+3+11/3+11/3+4+4+4+4)/8=(88/3)/8=11/3$

From this, we can calculate the zero-load latency.

$t_0=3*11/3+2*1+3=16$ cycles

1.c   Integrating Processors, Memories, and Networks

2 In-Order Superscalar Processors

2.a Pipeline Diagram for Single-Issue PARCv1 Processor

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
lw r1 , 0(r2)	F	D	X	M	W													
lw r3 , 0(r4)		F	D	X	M	W												
mul r1, r1, r6			F	D	X	M	W											
mul r3, r3, r7				F	D	X	M	W										
addu r8, r1, r3					F	D	X	M	W									
addu r9, r9, r8						F	D	X	M	W								
addiu r2, r2, 4							F	D	X	M	W							
addiu r4, r4, 4								F	D	X	M	W						
addiu r10, r10, -1									F	D	X	M	W					
bne r10, r0, loop										F	D	X	M	W				
opA											F	D	-	-	-			
opB												F	-	-	-	-		
lw r1 , 0(r2)													F	D	X	M	W	
lw r3 , 0(r4)														F	D	X	M	W

Figure 3: Pipeline Diagram for Single-Issue PARCv1 Processor

2.b Pipeline Diagram for Dual-Issue PARCv1 Processor

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lw r1 , 0(r2)	F	D	B0	B1	W												
lw r3 , 0(r4)	F	D	D	B0	B1	W											
mul r1, r1, r6		F	F	D	A0	A1	W										
mul r3, r3, r7		F	F	D	D	A0	A1	W									
addu r8, r1, r3				F	F	D	A0	A1	W								
addu r9, r9, r8				F	F	D	D	B0	B1	W							
addiu r2, r2, 4						F	F	D	A0	A1	W						
addiu r4, r4, 4						F	F	D	B0	B1	W						
addiu r10, r10, -1								F	D	B0	B1	W					
bne r10, r0, loop								F	D	D	A0	A1	W				
opA									F	F	D	-	-	-			
opB									F	F	D	-	-	-			
opC											F	-	-	-	-		
opD											F	-	-	-	-		
lw r1 , 0(r2)												F	D	B0	B1	W	
lw r3 , 0(r4)												F	D	D	B0	B1	W

Figure 4: Pipeline Diagram for Dual-Issue PARCv1 Processor

2.c Optimized Pipeline Diagram for Dual-Issue PARCv1 Processor

```
lw r1 , 0(r2)
addiu r2, r2, 4
lw r3 , 0(r4)
addiu r4, r4, 4
mul r1, r1, r6
addiu r10, r10, -1
mul r3, r3, r7
addu r8, r1, r3
addu r9, r9, r8
bne r10, r0, loop
```

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13
lw r1 , 0(r2)	F	D	B0	B1	W								
addiu r2, r2, 4	F	D	A0	A1	W								
lw r3 , 0(r4)		F	D	B0	B1	W							
addiu r4, r4, 4		F	D	A0	A1	W							
mul r1, r1, r6			F	D	A0	A1	W						
addiu r10, r10, -1			F	D	B0	B1	W						
mul r3, r3, r7				F	D	A0	A1	W					
addu r8, r1, r3				F	D	D	B0	B1	W				
addu r9, r9, r8					F	F	D	B0	B1	W			
bne r10, r0, loop					F	F	D	A0	A1	W			
opA							F	D	-	-	-		
opB							F	D	-	-	-		
opC								F	-	-	-	-	
opD								F	-	-	-	-	
lw r1 , 0(r2)									F	D	B0	B1	W
addiu r2, r2, 4									F	D	A0	A1	W

Figure 5: Pipeline Diagram for Dual-Issue PARCv1 Processor

4

3   Impact of Cache Access Time and Replacement Policy

3.a   Miss Rate Analysis

Transaction Address	tag	idx	m/h	L0	L1	L2	L3	L4	L5	L6	L7
0x024	0x0	0x2	m	-	-	-	-	-	-	-	-
0x030	0x0	0x3	m			0x0					
0x07c	0x0	0x7	m				0x0				
0x070	0x0	0x7	h								0x0
0x100	0x2	0x0	m								
0x110	0x2	0x1	m	0x2							
0x204	0x4	0x0	m		0x2						
0x214	0x4	0x1	m	0x4							
0x308	0x6	0x0	m		0x4						
0x110	0x2	0x1	m	0x6							
0x114	0x2	0x1	h		0x2						
0x118	0x2	0x1	h								
0x11c	0x2	0x1	h								
0x410	0x8	0x1	m								
0x110	0x2	0x1	m		0x8						
0x510	0xa	0x1	m		0x2						
0x110	0x2	0x1	m		0xa						
0x610	0xc	0x1	m		0x2						
0x110	0x2	0x1	m		0xc						
0x710	0xe	0x1	m		0x2						
Number of Misses = 16											
Miss Rate = 0.8											

Figure 6: Direct-Mapped Cache Contents Over Time

Transaction				Set 0		Set 1		Set 2		Set 3	
Address	tag	idx	m/h	Way 0	Way 1	Way 0	Way 1	Way 0	Way 1	Way 0	Way 1
0x024	0x0	0x2	m	-	-	-	-	-	-	-	-
0x030	0x0	0x3	m					0x0			
0x07c	0x1	0x3	m							0x0	
0x070	0x1	0x3	h								0x1
0x100	0x4	0x0	m								
0x110	0x4	0x1	m	0x4							
0x204	0x8	0x0	m			0x4					
0x214	0x8	0x1	m		0x8						
0x308	0xc	0x0	m				0x8				
0x110	0x4	0x1	h	0xc							
0x114	0x4	0x1	h								
0x118	0x4	0x1	h								
0x11c	0x4	0x1	h								
0x410	0x10	0x1	m								
0x110	0x4	0x1	h				0x10				
0x510	0x14	0x1	m								
0x110	0x4	0x1	h				0x14				
0x610	0x18	0x1	m								
0x110	0x4	0x1	h				0x18				
0x710	0x1c	0x1	m								
Number of Misses = 12											
Miss Rate = 0.6											

Figure 7: Two-Way Set-Associative Cache Contents Over Time with LRU Replacement

Transaction				Set 0		Set 1		Set 2		Set 3	
Address	tag	idx	m/h	Way 0	Way 1	Way 0	Way 1	Way 0	Way 1	Way 0	Way 1
0x024	0x0	0x2	m	-	-	-	-	-	-	-	-
0x030	0x0	0x3	m					0x0			
0x07c	0x1	0x3	m							0x0	
0x070	0x1	0x3	h								0x1
0x100	0x4	0x0	m								
0x110	0x4	0x1	m	0x4							
0x204	0x8	0x0	m			0x4					
0x214	0x8	0x1	m		0x8						
0x308	0xc	0x0	m				0x8				
0x110	0x4	0x1	h	0xc							
0x114	0x4	0x1	h								
0x118	0x4	0x1	h								
0x11c	0x4	0x1	h								
0x410	0x10	0x1	m								
0x110	0x4	0x1	h				0x10				
0x510	0x14	0x1	m								
0x110	0x4	0x1	m			0x14					
0x610	0x18	0x1	m								
0x110	0x4	0x1	m				0x18				
0x710	0x1c	0x1	m								
Number of Misses = 14											
Miss Rate = 0.7											

Figure 8: Two-Way Set-Associative Cache Contents Over Time with FIFO Replacement

3.b   Sequential Tag Check then Memory Access

Component	Delay Equation	Delay(τ)
addr_reg_M0	1	1
tag_decoder	$3 + 2\times2$	7
tag_mem	$10 + [(4+27)/16]$	12
tag_cmp	$3 + 2[\log2(26)]$	13
tag_and	2 - 1	1
data_decoder	$3 + 2\times3$	9
data_mem	$10 + [(8+128)/16]$	19
rdata_mux	$3[\log2(4)] + [32/8]$	10
rdata_reg_M1	1	1
Total		**73**
addr_reg_M0	1	1
tag_decoder	$3 + 2\times2$	7
tag_mem	$10 + [(4+27)/16]$	12
tag_cmp	$3 + 2[\log2(26)]$	13
tag_and	2 - 1	1
data_decoder	$3 + 2\times3$	9
data_mem	$10 + [(8+128)/16]$	19
Total		**62**

Figure 9: Critical Path and Cycle Time for 2-Way Set-Associative Cache with
Serialized Tag Check before Data Access

The reason that the 2-way set-associative microarchitecture is slower than the direct-mapped microarchitecture is the need for the tag check result to go through the data_decoder. It happens that the data_decoder?s delay is relatively significant (9τ). This connection is needed so that the data can be outputted from the correct way.

3.c   Parallel Read Hit Path

Component	Delay Equation	Delay(τ)
addr_reg_M0	1	1
addr_mux	$3[\log2(2)] + [5/8]$	4
data_decoder	$3 + 2\times3$	9
data_mem	$10 + [(8+128)/16]$	19
rdata_mux	$3[\log2(4)] + [32/8]$	10
rdata_reg_M1	1	1
Total		**44**

Figure 10: Critical Path and Cycle Time for Direct Mapped Cache with
Parallel Read Hit

Component	Delay Equation	Delay(τ)
addr_reg_M0	1	1
addr_mux	$3[\log2(2)] + [5/8]$	4
data_decoder	$3 + 2\times2$	7
data_mem	$10 + [(8+128)/16]$	19
rdata_mux	$3[\log2(4)] + [32/8]$	10
way_mux	$3[\log2(2)] + [32/8]$	7
rdata_reg_M1	1	1
Total		**49**

Figure 11: Critical Path and Cycle Time for 2-Way Set-Associative Cache with
Parallel Read Hit

The reason that the 2-way set-associative microarchitecture is slower than the direct-mapped microarchitecture is the way_mux, which is needed to output the data from the correct way. This mux has a delay of 7τ, which is relatively significant.

3.d   Pipelined Write Hit Path

Component	Delay Equation	Delay(τ)
addr_reg_M0	1	1
tag_decoder	$3 + 2\times3$	9
tag_mem	$10 + [(8+26)/16]$	13
tag_cmp	$3 + 2[\log2(25)]$	13
tag_and	2 - 1	1
wen_and	2 - 1	1
wen_reg_M1	1	1
Total		**39**

Figure 12: Critical Path and Cycle Time for Direct Mapped Cache with
Pipelined Write Hit

Component	Delay Equation	Delay(τ)
addr_reg_M0	1	1
tag_decoder	$3 + 2\times2$	7
tag_mem	$10 + [(4+27)/16]$	12
tag_cmp	$3 + 2[\log2(25)]$	13
tag_and	2 - 1	1
wen_and	2 - 1	1
wen_reg_M1	1	1
Total		**36**

Figure 13: Critical Path and Cycle Time for 2-Way Set-Associative Cache with
Pipelined Write Hit

The reason that the direct-mapped microarchitecture is slower than the 2-way set-associative microarchitecture is the larger tag_decoder, which is needed to fetch the correct tag from the tag memory. Because the direct mapped cache uses a 3 to 8 decoder instead of a smaller 2 to 4 decoder of the 2-way set associative cache, the critical path is longer. There is also 1τ extra delay for the tag_mem due to it being 8 rows instead of 4 rows in the 2-way set-associative cache.

3.e   Average Memory Access Latency

Associativity	μarch	Replacement Policy	Hit Time (τ)	Miss Rate (ratio)	Miss Penalty (τ)	AMAL (τ)
Direct Mapped	Seq	n/a	68	0.8	300	308
2-way Set Assoc	Seq	LRU	73	0.6	300	253
2-way Set Assoc	Seq	FIFO	73	0.7	300	283
Direct Mapped	PP	n/a	44	0.8	300	284
2-way Set Assoc	PP	LRU	49	0.6	300	229
2-way Set Assoc	PP	FIFO	49	0.7	300	259

Figure 14: Average Memory Access Latency for Six Cache Configurations

The pipelined 2-way set-associative cache with a LRU replacement policy has the lowest average memory access time. I think that this conclusion is fairly general due to several factors. Set associative caches generally performed better

than direct mapped caches due to its better ability to handle sparser data. Due to the pipelined write and parallel read hit, the cycle time is also much lower than the sequential architecture. The LRU policy is very beneficial for real world programs that generally display temporal locality in data use. I think that in most cases, it is safe to say that we should choose this configuration. However, in some specific cases, this may not be the best choice. For example, for programs that have very structured spatial locality, it might be better to use a direct mapped cache due to its larger index. This can improve performance slightly with the lower cycle time. Also in technology where energy consumption and heat dissipation is a larger concern, the parallel read hit design can be bad due to wasted power in reading the data memory even though there was no tag hit.

4   Array vs. List Cache Behavior

| | Number of | | Execution | CPI Breakdown | | | |
Part	Instructions	CPI	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 15: 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 16: 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:	1	2	3	4	m 5	m 6	m 7	m 8	u 9	m 10	m 11	m 12	m 13	u 14	u 15	u 16	u 17	u 18	u 19	u 20	c 21	c 22	m 23	m 24	m 25	m 26	u 27	m 28	m 29	m 30	m 31	u 32	u 33	u 34	u 35	u 36	u 37	u 38	c 39	c 40	u 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 17: 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.