

ECE 4750 PSET 2

Tim Yao (ty252)

Oct 9, 2015

1 PARCv1 Instruction Cache

1.a Categorizing Cache Misses

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

Figure 1: Cache Miss Type

1.b Average Memory Access Latency

Looking at iteration 2, we can see that there are 2 misses out of the 5 instructions. Therefore the miss rate for 64 iterations of the loop is 0.4.

The average memory access latency is:

$$\text{AMAL} = (\text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Penalty})$$

$$\text{AMAL} = 1 + (0.4 \times 5)$$

$$\text{AMAL} = 3 \text{ cycles}$$

The AMAL is dominated by conflict misses, as shown by the miss chart above. Compulsory misses only occur on the first iteration of the loop.

1.c Set-Associativity

The cache performance will increase significantly, because there will no longer be conflict misses during the loop. With this new cache microarchitecture, only compulsory misses will be left.

2 Page-Based Memory Translation

2.a Two-Level Page Tables

The 16-bit virtual address is used as the following:

Bits 14-15	Bits 12-13	Bits 0-11
XX	XX	XXXXXXXXXXXXXX
Virtual Page Number		Page Offset
L1 Index	L2 Index	Page Offset

Figure 2: Virtual Address Usage

Page Tables:

Paddr of PTE	Page-Table Entry	
	Valid	Paddr
0xffffc	1	0xfffe0
0xffff8		
0xffff4		
0xffff0	1	0xfffb0
0xfffec	1	0x05000
0xfffe8	1	0x07000
0xfffe4		
0xfffe0		
0xfffdc		
0xfffd8		
0xfffd4		
0xfffd0		
0xfffcc		
0xfffc8		
0xfffc4		
0xfffc0		
0xffbbc	1	0x01000
0xfffb8	1	0x04000
0xfffb4	1	0x00000
0xfffb0		

Figure 3: Contents of Physical Memory with Page Tables

2.b Translation-Lookaside Buffer

Transaction Address	VPN	Page Offset	m/h	Total Num Mem Accesses	Way 0		Way 1	
					VPN	PPN	VPN	PPN
0xeff4	0xe	0xff4	m	3	-	-	-	-
0x2ff0	0x2	0xff0	m	3	0xe	0x07		
0xeff8	0xe	0xff8	h	1			0x2	0x04
0x2ff4	0x2	0xff4	h	1				
0xeffc	0xe	0xffc	h	1				
0x2ff8	0x2	0xff8	h	1				
0xf000	0xf	0x000	m	3				
0x2ffc	0x2	0xffc	h	1	0xf	0x05		
0xf004	0xf	0x004	h	1				
0x3000	0x3	0x000	m	3				
0xf008	0xf	0x008	h	1			0x3	0x01
0x3004	0x3	0x004	h	1			0x2	0x04
0xf00c	0xf	0x00c	h	1				
0x3008	0x3	0x008	h	1				

Figure 4: TLB Contents Over Time

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 5: 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 6: 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 7: 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	$3 + 2 \times 2$	7
tag_mem	$10 + \lfloor (4+27)/16 \rfloor$	12
tag_cmp	$3 + 2\lceil \log_2(26) \rceil$	13
tag_and	$2 - 1$	1
data_decoder	$3 + 2 \times 3$	9
data_mem	$10 + \lfloor (8+128)/16 \rfloor$	19
rdata_mux	$3\lceil \log_2(4) \rceil + \lfloor 32/8 \rfloor$	10
rdata_reg_M1	1	1
Total		73
addr_reg_M0	1	1
tag_decoder	$3 + 2 \times 2$	7
tag_mem	$10 + \lfloor (4+27)/16 \rfloor$	12
tag_cmp	$3 + 2\lceil \log_2(26) \rceil$	13
tag_and	$2 - 1$	1
data_decoder	$3 + 2 \times 3$	9
data_mem	$10 + \lfloor (8+128)/16 \rfloor$	19
Total		62

Figure 8: 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	$3\lceil \log_2(2) \rceil + \lfloor 5/8 \rfloor$	4
data_decoder	$3 + 2 \times 3$	9
data_mem	$10 + \lfloor (8+128)/16 \rfloor$	19
rdata_mux	$3\lceil \log_2(4) \rceil + \lfloor 32/8 \rfloor$	10
rdata_reg_M1	1	1
Total		44

Figure 9: 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	$3\lceil \log_2(2) \rceil + \lfloor 5/8 \rfloor$	4
data_decoder	$3 + 2 \times 2$	7
data_mem	$10 + \lfloor (8+128)/16 \rfloor$	19
rdata_mux	$3\lceil \log_2(4) \rceil + \lfloor 32/8 \rfloor$	10
way_mux	$3\lceil \log_2(2) \rceil + \lfloor 32/8 \rfloor$	7
rdata_reg_M1	1	1
Total		49

Figure 10: 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	$3 + 2 \times 3$	9
tag_mem	$10 + \lceil (8+26)/16 \rceil$	13
tag_cmp	$3 + 2\lceil \log_2(25) \rceil$	13
tag_and	$2 - 1$	1
wen_and	$2 - 1$	1
wen_reg_M1	1	1
Total		39

Figure 11: 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	$3 + 2 \times 2$	7
tag_mem	$10 + \lceil (4+27)/16 \rceil$	12
tag_cmp	$3 + 2\lceil \log_2(25) \rceil$	13
tag_and	$2 - 1$	1
wen_and	$2 - 1$	1
wen_reg_M1	1	1
Total		36

Figure 12: 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	μ arch	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 13: 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

Part	Number of Instructions	CPI	Execution Time (cyc)	CPI Breakdown			
				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 = $8 \times 8 + 24 \times 8 = 256$ cycles, $256/256 = 1.00$ CPI

m = $8 \times 8 + 24 \times 0 = 64$ cycles, $64/256 = 0.25$ CPI

r = $8 \times 0 + 24 \times 0 = 0$ cycles, $0/256 = 0.00$ CPI

c = $8 \times 2 + 24 \times 2 = 64$ 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	c	c	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	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														
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	c	c		m	m	m	m	u	m	m	m	m	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	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	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.