CS152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Spring 2018

SOLUTIONS

Caches and the Memory Hierarchy

Assigned February 8 Problem Set #2 *Due Wed, February 21*

http://inst.eecs.berkeley.edu/~cs152/sp18

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in his/her own solution to the problems.

The problem sets also provide essential background material for the exams. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets you are unlikely to succeed at the exams! Homework assignments are due at the beginning of class on the due date. Late homework will not be accepted.

Problem 1: Cache Access-Time & Performance

Here is the completed Table 2.1-1 for 2.1.A and 2.1.B.

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	20×(# of index bits) + 100	Tag	340	300
		Data	340	300
Memory array	$20\times \log_2$ (# of rows) +	Tag	422	425
	$20\times \log_2$ (# of bits in a row) + 100	Data	500	500
Comparator	20×(# of tag bits) + 100		400	440
N-to-1 MUX	$50\times\log_2 N + 100$		250	250
Buffer driver	200			200
Data output driver	50×(associativity) + 100		150	300
Valid output driver	100		100	100

Problem 1.A **Access Time: Direct-Mapped**

To use the delay equations, we need to know how many bits are in the tag and how many are in the index. We are given that the cache is addressed by word, and that input addresses are 32-bit byte addresses; the two low bits of the address are not used.

Since there are 8 (2^3) words in the cache line, *3 bits are needed to select the correct word from the cache line.*

In a 128 KB direct-mapped cache with 8 word (32 byte) cache lines, *there are $4\times2^{10} = 2^{12}$ cache lines (128KB/32B)*. 12 bits are needed to address 2^{12} cache lines, so *the number of index bits is 12. The remaining 15 bits (32 – 2 – 3 – 12) are the tag bits.*

We also need the number of rows and the number of bits in a row in the tag and data memories. The number of rows is simply the number of cache lines (2^{12}), which is the same for both the tag and the data memory. The number of bits in a row for the tag memory is the sum of the number of tag bits (15) and the number of status bits (2), 17 bits total. The number of bits in a row for the data memory is the number of bits in a cache line, which is 256 (32 bytes × 8 bits/byte).

With 8 words in the cache line, we need an 8-to-1 MUX. Since there is only one data output driver, its associativity is 1.

Decoder (Tag) = 20 × (# of index bits) + 100 = 20 × 12 + 100 = 340 ps
Decoder (Data) = 20 × (# of index bits) + 100 = 20 × 12 + 100 = 340 ps

Memory array (Tag) = 20 × log₂(# of rows) + 20 × log₂(# bits in a row) + 100
= $20 \times \log_2(2^{12}) + 20 \times \log_2(17) + 100 \approx 422$ ps
Memory array (Data) = 20 × log₂(# of rows) + 20 × log₂(# bits in a row) + 100

$= 20 \times \log_2(2^{12}) + 20 \times \log_2(256) + 100 = 500$ ps

Comparator $= 20 \times$ (# of tag bits) $+ 100 = 20 \times 15 + 100 = 400$ ps

N-to-1 MUX $= 50 \times \log_2(N) + 100 = 50 \times \log_2(8) + 100 = 250$ ps

Data output driver $= 50 \times$ (associativity) $+ 100 = 500 \times 1 + 100 = 150$ ps

To determine the critical path for a cache read, we need to compute the time it takes to go through each path in hardware, and find the maximum.

Time to tag output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time) + (valid output driver time)
$\approx 340 + 422 + 400 + 50 + 100 = 1312$ ps

Time to data output driver
= (data decode time) + (data memory access time) + (mux time) + (data output driver time)
$= 340 + 500 + 250 + 150 = 1240$ ps

The critical path is therefore the tag read going through the comparator. The access time is 1312 ps. At 1.5 GHz, it takes 1.312×1.5, or 2 cycles, to do a cache access.

Problem 1.B — Access Time: Set-Associative

As in 2.1.A, the low two bits of the address are not used, and 3 bits are needed to select the appropriate word from a cache line. However, now we have a 128 KB 4-way set associative cache. Since each way is 32 KB and cache lines are 32 bytes, *there are 2^{10} lines in a way (32KB/32B) that are addressed by 10 index bits. The number of tag bits is then $(32 – 2 – 3 – 10)$, or 17.*

The number of rows in the tag and data memory is 2^{10}, or the number of sets. The number of bits in a row for the tag memory is now quadruple the sum of the number of tag bits (17) and the number of status bits (2), 76 bits total. The number of bits in a row for the data memory is twice the number of bits in a cache line, which is 1024 (4×32 bytes $\times 8$ bits/byte).

As in 1.A, we need an 8-to-1 MUX. However, since there are now four data output drivers, the associativity is 4.

Decoder (Tag) $= 20 \times$ (# of index bits) $+ 100 = 20 \times 10 + 100 = 300$ ps
Decoder (Data) $= 20 \times$ (# of index bits) $+ 100 = 20 \times 10 + 100 = 300$ ps

Memory array (Tag) $= 20 \times \log_2$(# of rows) $+ 20 \times \log_2$(# bits in a row) $+ 100$
$= 20 \times \log_2(2^{10}) + 20 \times \log_2(76) + 100 \approx 425$ ps

Memory array (Data) = 20 × log$_2$(# of rows) + 20 × log$_2$(# bits in a row) + 100
= 20 × log$_2$(2^{10}) + 20 × log$_2$(1024) + 100 = 500 ps

Comparator = 20 × (# of tag bits) + 100 = 20 × 17 + 100 = 440 ps

N-to-1 MUX = 50 × log$_2$(N) + 100 = 50 × log$_2$(8) + 100 = 250 ps

Data output driver = 50 × (associativity) + 100 = 50 × 4 + 100 = 300 ps

Time to valid output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time) + (OR gate time) + (valid output driver time)
= 300 + 425 + 440 + 50 + 100 + 100 = 1415 ps

There are two paths to the data output drivers, one from the tag side, and one from the data side. Either may determine the critical path to the data output drivers.

Time to get through data output driver through tag side
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time) + (buffer driver time) + (data output driver)
= 300 + 425 + 440 + 50 + 200 + 300 = 1715 ps

Time to get through data output driver through data side
= (data decode time) + (data memory access time) + (mux time) + (data output driver)
= 300 + 500 + 250 + 300 = 1350 ps

From the above calculations, it's clear that the critical path leading to the data output driver goes through the tag side.

The critical path for a read therefore goes through the tag side comparators, then through the buffer and data output drivers. ***The access time is 1715 ps***. The main reason that the 4-way set associative cache is slower than the direct-mapped cache is that the data output drivers need the results of the tag comparison to determine which, if either, of the data output drivers should be putting a value on the bus. ***At 1.5 GHz, it takes 1.715 × 1.5, or 3 cycles, to do a cache access.***

It is important to note that the structure of cache we've presented here does not describe all the details necessary to operate the cache correctly. There are additional bits necessary in the cache which keep track of the order in which lines in a set have been accessed (for replacement). We've omitted this detail for sake of clarity.

Problem 1.C **Miss-rate analysis**

For the direct-mapped cache:

Addr = 12 bits Addr[11:0]
tag = 5 bits Addr[11:7] (12bits - index_sz - offset_sz)
index = 3 bits Addr[6:4] (2^3 = 8 lines)
offset= 4 bits Addr[3:0] (2^4 = 16 bytes/line)

D-map									
	line in cache (tag)								hit?
Address	L0	L1	L2	L3	L4	L5	L6	L7	
110	inv	2	inv	inv	inv	inv	inv	inv	no
136				2					no
202	4								no
1A3			3						no
102	2								no
361							6		no
204	4								no
114									yes
1A4									yes
177								2	no
301	6								no
206	4								no
135									yes

	D-map
Total Misses	10
Total Accesses	13

For the 4-way set associative cache, there are now 2 sets and 4 ways:

Addr = 12 bits Addr[11:0]
tag = 7 bits Addr[11:5] (12bits - index_sz - offset_sz)
index = 1 bits Addr[4:4] (2^1 = 2 sets)
offset= 4 bits Addr[3:0] (2^4 = 16 bytes/line)

4-way Address	LRU								
	line in cache								hit?
	Set 0				Set 1				
	way0	**way1**	**Way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	inv	inv	inv	8	inv	inv	inv	no
136						9			no
202	10								no
1A3		0D							no
102			08						no
361				1B					no
204	10								yes
114					08				yes
1A4		0D							yes
177							0B		no
301			18						no
206	10								yes
135						09			yes

	4-way LRU
Total Misses	8
Total Accesses	13

4-way	FIFO								
	line in cache (tag)								hit?
Address	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	inv	inv	inv	8	inv	inv	inv	no
136						9			no
202	10								no
1A3		0D							no
102			08						no
361				1B					no
204	10								yes
114					08				yes
1A4		0D							yes
177							0B		no
301	18								no
206		10							no
135						09			yes

	4-way FIFO
Total Misses	9
Total Accesses	13

Problem 1.D Average Latency

The miss rate for the direct-mapped cache is 10/13. The miss rate for the 4-way LRU set associative cache is 8/13.

The average memory access latency is (hit time) + (miss rate) × (miss time).

For the direct-mapped cache, the average memory access latency would be (2 cycles) + (10/13) × (20 cycles) = 17.38 ≈ 18 cycles.

For the LRU set associative cache, the average memory access latency would be (3 cycles) + (8/13) × (20 cycles) = 15.31 ≈ 16 cycles.

The set associative cache is better in terms of average memory access latency.

For the above example, LRU has a slightly smaller miss rate than FIFO. This is because the FIFO policy replaced the {20} block instead of the {10} block during the 12th access, because the {20} block has been in the cache longer even though the {10} was least recently used, whereas ***the LRU policy took advantage of temporal/spatial locality***.

LRU does not always outperforms FIFO. Assume we have a fully-associative cache with two lines and an access sequence for 1, 2, 1, 3, 2. There is a miss with LRU for the last access while there is a hit with FIFO.

Problem 2: Loop Ordering

Problem 2.A

Each element of the matrix can only be mapped to a particular cache location because the cache here is a direct-mapped data cache. Matrix A has 64 columns and 128 rows. Since each row of matrix has 64 32-bit integers and each cache line can hold 8 words, each row of the matrix fits exactly into eight (64/8) cache lines as the following:

0	A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]	A[0][5]	A[0][6]	A[0][7]
1	A[0][8]	A[0][9]	A[0][10]	A[0][11]	A[0][12]	A[0][13]	A[0][14]	A[0][15]
2	A[0][16]	A[0][17]	A[0][18]	A[0][19]	A[0][20]	A[0][21]	A[0][22]	A[0][23]
3	A[0][24]	A[0][25]	A[0][26]	A[0][27]	A[0][28]	A[0][29]	A[0][30]	A[0][31]
4	A[0][32]	A[0][33]	A[0][34]	A[0][35]	A[0][36]	A[0][37]	A[0][38]	A[0][39]
5	A[0][40]	A[0][41]	A[0][42]	A[0][43]	A[0][44]	A[0][45]	A[0][46]	A[0][47]
6	A[0][48]	A[0][49]	A[0][50]	A[0][51]	A[0][52]	A[0][53]	A[0][54]	A[0][55]
7	A[0][56]	A[0][57]	A[0][58]	A[0][59]	A[0][60]	A[0][61]	A[0][62]	A[0][63]
8	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]	A[1][5]	A[1][6]	A[1][7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Loop A accesses memory sequentially (each iteration of Loop A sums a row in matrix A), an access to a word that maps to the first word in a cache line will miss but the next seven accesses will hit. Therefore, Loop A will only have compulsory misses (128×(64/8) or 1024 misses).

The consecutive accesses in Loop B will use every eighth cache line (each iteration of Loop B sums a column in matrix A). Fitting one column of matrix A, we would need 128×8 or 1024 cache lines. However, our 4KB data cache with 32B cache line only has 128 cache lines. When Loop B accesses a column, all the data that the previous iteration might have brought in would have already been evicted. Thus, every access will cause a cache miss (64×128 or 8192 misses).

The number of cache misses for Loop A:________1024___________

The number of cache misses for Loop B:________8192___________

Problem 2.B

Since *Loop* A accesses memory sequentially, we can overwrite the cache lines that were previous brought in. Loop A will only require 1 cache line to run without any cache misses other than compulsory misses.

For Loop B to run without any cache misses other than compulsory misses, the data cache needs to have the capacity to hold one column of matrix A. Since the consecutive accesses in Loop B will use every eighth cache line and we have 128 elements in a matrix A column, Loop B requires 128×8 or 1024 cache lines.

Data-cache size required for Loop A: _________1__________ cache line(s)

Data-cache size required for Loop B: _______1024_________ cache line(s)

Problem 2.C

Loop A still only has compulsory misses (128×(64/8) or 1024 misses).

Because of the fully-associative data cache, Loop B now can fully utilize the cache and the consecutive accesses in Loop B will no longer use every eighth cache line. Fitting one column of matrix A, we now would only need 128 cache lines. Since 4KB data cache with 8-word cache lines has 128 cache lines, Loop B only has compulsory misses (128×(64/8) or 1024 misses).

The number of cache misses for Loop A:______1024_________

The number of cache misses for Loop B:______1024_________

Problem 3: Microtagged Cache

Problem 3.A Cache Cycle Time

Component	Delay equation (ps)		Baseline	Microtagged
Decoder	20×(# of index bits) + 100	Tag	240	240
		Data	240	240
Memory array	$20 \times \log_2$ (# of rows) + $20 \times \log_2$ (# of bits in a row) + 100	Tag	330	330
		Data	440	440
		Microtag		306 (tag 8 bits+ valid 1bit)
Comparator	20×(# of tag bits) + 100	Tag	500	500
		Microtag		260
N-to-1 MUX	$50 \times \log_2 N + 100$		250	250
Buffer driver	200		200	200
Data output driver	50×(associativity) + 100		300	300
Valid output driver	100		100	100

What is the old critical path? The old cycle time (in ps)?

critical path is through tag check (although you should verify this by also computing path through data read)

tag decoder → tag read → comparator → 2-in AND → buffer driver → data output driver

240 + 330 + 500 + 50 + 200 + 300 = 1620 ps, or 1.62 ns

What is the new critical path? The new cycle time (in ps)?

What is the new critical path?

tag check = 1320 ps (-500 since we no longer drive data out, but +200 for OR and valid driver)

microtag to buffer driver:
microtag decoder → utag array → utag comparator → 2-input AND → buffer driver
240 + 306 + 260 + 50 + 200 = 1056 ps

data up to data output driver (see which arrives first, microtags or data)
data decoder → data array → N-1 MUX
240 + 440 + 250 = 930 ps

So microtag takes longer than data to reach output drivers, so the path along microtag to data out is 1056 ps + 300 ps = 1356 ps

This path is longer than the full tag check which can occur in parallel which takes
1320 ps.

final answer: 1350 ps, or 1.356 ns.

Problem 3.B AMAT

AMAT = hit_time + miss_rate*miss_penalty = X + (0.05)(20ns) = X + 1ns, where X is the hit time calculated from 3.A

AMAT_old = 1.62 + 1 = 2.62 ns

AMAT_new = 1.356 + 1 = 2.356 ns

Problem 3.C **Constraints**

Increases conflict misses.

it will be worse than a 4-way set-associative cache because there are additional constraints (and thus more conflict misses), but it will be at least as good as a direct-mapped cache because two cache lines that have the same microtag and set bits would also index into the same line (thus the same constrained behavior occurs in both direct-mapped and microtagged caches), but increasing associativity with the microtagged cache would decrease conflict misses for lines that don't share the same microtag bits.

Problem 4: Victim Cache Evaluation

Problem 4.A Baseline Cache Design

Component	Delay equation (ps)	FA(ps)
Comparator	20×(# of tag bits) + 100	680
N-to-1 MUX	$50\times\log_2 N + 100$	150
Buffer driver	200	200
AND gate	100	100
OR gate	$50\times \log_2 N + 100$	200
Data output driver	50×(associativity) + 100	300
Valid output driver	100	100

The Input Address has 32 bits. The bottom two bits are discarded (cache is word-addressable) and bit 2 is used to select a word in the cache line. Thus the Tag has 29 bits. The Tag+Status line in the cache is 31 bits.

The MUXes are 2-to-1, thus N is 2. The associativity of the Data Output Driver is 4 – there are four drivers driving each line on the common Data Bus.

Delay to the Valid Bit is equal to the delay through the Comperator, AND gate, OR gate, and Valid Output Driver. Thus it is 680 + 100 + 200 + 100 = 1080 ps.

Delay to the Data Bus is delay through MAX ((Comperator, AND gate, Buffer Driver), (MUX)), Data Output Drivers. Thus it is MAX (680 + 100 + 200, 150) + 300 = MAX (980, 150) + 300 = 980 + 300 = 1280 ps.

Critical Path Cache Delay: 1280 ps

Problem 4.B Victim Cache Behavior

Input Address	Main Cache (tag)									Victim Cache (tag)		
	L0	L1	L2	L3	L4	L5	L6	L7	Hit?	Way0	Way1	Hit?
	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv	-
0	0								N			N
80	1								N	0		N
4	0								N	8		Y
A0			1						N			N
10		0							N			N
C0					1				N			N

18		0							Y			
20			0						N		A	N
8C	1								N	0		Y
28			0						Y			
AC			1						N		2	Y
38				0					N			N
C4					1				Y			
3C				0					Y			
48					0				N	C		N
0C	0								N		8	N
24			0						N	A		N

Problem 4.C — Average Memory Access Time

15% of accesses will take 50 cycles less to complete, so the average memory access improvement is 0.15 * 50 = 7.5 cycles.

Problem 5: Three C's of Cache Misses

Mark whether the following modifications will cause each of the categories to **increase, decrease**, or whether the modification will have **no effect**. You can assume the baseline cache is set associative. **Explain your reasoning**.

	Compulsory Misses	Conflict Misses	Capacity Misses
Double the associativity (capacity and line size constant) halves # of sets	No effect If the data wasn't ever in the cache, increasing associativity with the constraints won't change that.	Decrease Typically higher associativity reduces conflict misses because there are more places to put the same element.	No effect Capacity was given as a constant.
Halving the line size (associativity and # sets constant) halves capacity	Increase Shorter lines mean less "prefetching" for shorter lines. It reduces the cache's ability to exploit spatial locality.	Increase There are misses that can be removed by increasing the associativity.	Increase Capacity has been cut in half.
Doubling the number of sets (capacity and line size constant) halves associativity	No effect If the data wasn't ever in the cache, increasing the number of sets with the constraints won't change that.	Increase Less associativity.	No effect Capacity is still constant

	Compulsory Misses	Conflict Misses	Capacity Misses
Adding prefetching	Decreases hopefully a good prefetcher can bring data in before we use it (either software prefetch inst or a prefetch unit that recognizes a particular access pattern)	Increase prefetch data could pollute the cache No effect with stream buffers	Increase prefetch data could possibly pollute the cache No effect with stream buffers

Problem 6: Memory Hierarchy Performance

Mark whether the following modifications will cause each of the categories to **increase, decrease**, or whether the modification will have **no effect**. You can assume the baseline cache is set associative. **Explain your reasoning**.

	Hit Time	Miss Rate	Miss Penalty
Double the associativity (capacity and line size constant) halves # of sets	Increases sets decrease, so tag gets larger. Also more tags must be checked, and more ways have to be muxed outs	Decrease less conflict misses	no effect this is dominated by the outer memory hierarchy

Halving the line size (associativity and # sets constant) halves capacity	Decreases the cache becomes physically smaller, so this probably dominates the increased tag check time (tag grows by 1 bit)	Increases smaller capacity, less spatial locality (more compulsory misses)	Decreases uses less bandwidth out to memory
Doubling the number of sets (capacity and line size constant) halves associativity	Decreases halved the associativity means less logic getting data out of the ways, tag is also smaller	Increases increases conflict misses because associativity gets halved	no effect this is dominated by the outer memory hierarchy
Adding prefetching	No effect prefetching isn't on the hit path	Decreases the purpose of a prefetcher is to reduce the miss rate by bringing in data ahead of time (In general, decrease in compulsory misses > increase in capacity and conflict misses. Otherwise, there is no reason to use prefetching)	No effect in general because prefetching dose not interfere with cache miss handling. May increase due to bandwidth pollution but we can(should) give a priority on cache misses over prefetch requests. Increase with stream buffers because stream buffers should be examined first before sending requests to the outer memory. May decrease because a prefetch can be inflight when a miss occurs (but this is unlikely).