CS152
Computer Architecture and Engineering

February 16, 2011

Caches and the Memory Hierarchy

Assigned February 16 Problem Set #2 *Due March 2*

---

http://inst.eecs.berkeley.edu/~cs152/sp11

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 own solution to the problems.

The problem sets also provide essential background material for the quizzes. 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 quizzes! We will distribute solutions to the problem sets on the day the problem sets are due to give you feedback. Homework assignments are due at the beginning of class on the due date. Late homework will not be accepted.

Problem 2.1: Cache Access-Time & Performance

This problem requires the knowledge of Handout #2 (Cache Implementations) and Lectures 6 & 7. Please, read these materials before answering the following questions.

Ben is trying to determine the best cache configuration for a new processor. He knows how to build two kinds of caches: direct-mapped caches and 4-way set-associative caches. The goal is to find the better cache configuration with the given building blocks. He wants to know how these two different configurations affect the clock speed and the cache miss-rate, and choose the one that provides better performance in terms of average latency for a load.

Problem 2.1.A — Access Time: Direct-Mapped

Now we want to compute the access time of a direct-mapped cache. We use the implementation shown in Figure H2-A in Handout #2. Assume a 128-KB cache with 8-word (32-byte) cache lines. The address is 32 bits and byte-addressed, so the two least significant bits of the address are ignored since a cache access is word-aligned. The data output is also 32 bits (1 word), and the MUX selects one word out of the eight words in a cache line. Using the delay equations given in Table 2.1-1, **fill in the column for the direct-mapped (DM) cache in the table**. *In the equation for the data output driver, 'associativity' refers to the associativity of the cache (1 for direct-mapped caches, A for A-way set-associative caches).*

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	200×(# of index bits) + 1000	Tag		
		Data		
Memory array	200×$\log_2$ (# of rows) + 200×$\log_2$ (# of bits in a row) + 1000	Tag		
		Data		
Comparator	200×(# of tag bits) + 1000			
N-to-1 MUX	500×$\log_2$ N + 1000			
Buffer driver	2000			
Data output driver	500×(associativity) + 1000			
Valid output driver	1000			

Table 2.1-1: Delay of each Cache Component

What is the critical path of this direct-mapped cache for a cache read? What is the access time of the cache (the delay of the critical path)? To compute the access time, assume that a 2-input gate (AND, OR) delay is 500 ps. If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?

We also want to investigate the access time of a set-associative cache using the 4-way set-associative cache in Figure H2-B in Handout #2. Assume the total cache size is still 128-KB (each way is 32-KB), a 4-input gate delay is 1000 ps, and all other parameters (such as the input address, cache line, etc.) are the same as part 2.1.A. **Compute the delay of each component, and fill in the column for a 4-way set-associative cache in Table 2.1-1.**

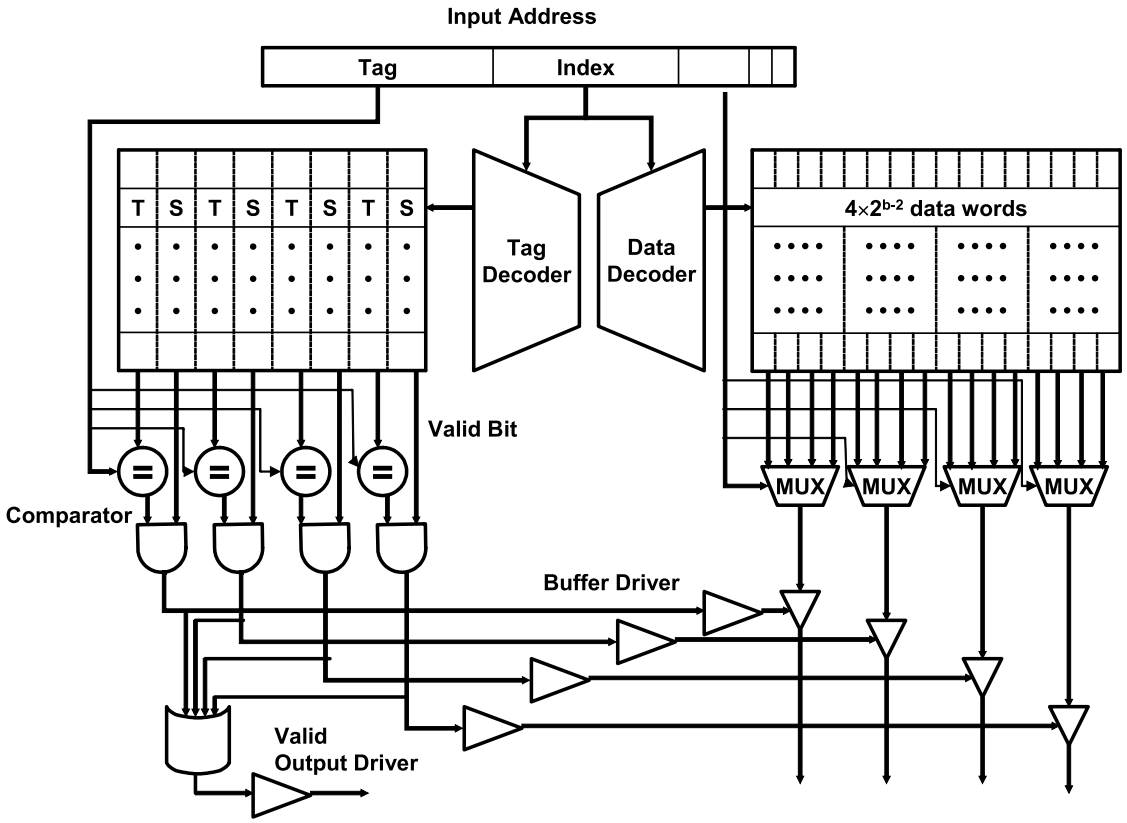

Figure 2.1: 4-way set-associative cache

What is the critical path of the 4-way set-associative cache? What is the access time of the cache (the delay of the critical path)? What is the main reason that the 4-way set-associative cache is slower than the direct-mapped cache? If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?

Problem 2.1.C **Miss-rate analysis**

Now Ben is studying the effect of set-associativity on the cache performance. Since he now knows the access time of each configuration, he wants to know the miss-rate of each one. For the miss-rate analysis, Ben is considering two small caches: a direct-mapped cache with 8 lines with 16 bytes/line, and a 4-way set-associative cache of the same size (i.e., both caches are 128 bytes). For the set-associative cache, Ben tries out two replacement policies – least recently used (LRU) and round robin (FIFO).

Ben tests the cache by accessing the following sequence of hexadecimal byte addresses, starting with empty caches. For simplicity, assume that the addresses are only 12 bits. **Complete the following tables** for the direct-mapped cache and both types of 4-way set-associative caches showing the progression of cache contents as accesses occur (in the tables, 'inv' = invalid, and the column of a particular cache line contains the {tag,index} contents of that line). Also, for each address calculate the *tag* and *index* (which should help in filling out the table). *You only need to fill in elements in the table when a value changes.*

D-map											
Address	**tag**	**index**	line in cache								hit?
			L0	L1	L2	L3	L4	L5	L6	L7	
110			inv	11	inv	inv	inv	inv	inv	inv	no
136						13					no
202			20								no
1A3											
102											
361											
204											
114											
1A4											
177											
301											
206											
135											

	D-map
Total Misses	
Total Accesses	

4-way **Address**	**tag**	**index**	line in cache								LRU hit?
			Set 0				Set 1				
			way0	**way1**	**Way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110			inv	Inv	Inv	inv	11	inv	inv	inv	no
136							11	13			no
202			20								no
1A3											
102											
361											
204											
114											
1A4											
177											
301											
206											
135											

	4-way LRU
Total Misses	
Total Accesses	

4-way **Address**	**tag**	**index**	line in cache								FIFO hit?
			Set 0				Set 1				
			way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110			inv	Inv	Inv	inv	11	inv	inv	inv	no
136								13			no
202			20								no
1A3											
102											
361											
204											
114											
1A4											
177											
301											
206											
135											

	4-way FIFO
Total Misses	
Total Accesses	

Problem 2.1.D **Average Latency**

Assume that the results of the above analysis can represent the average miss-rates of the direct-mapped and the 4-way LRU 128-KB caches studied in 2.1.A and 2.1.B. What would be the average memory access latency in CPU cycles for each cache (assume that the cache miss penalty is 20 cycles)? Which one is better? For the different replacement policies for the set-associative cache, which one has a smaller cache miss rate for the address stream in 2.1.C? Explain why. Is that replacement policy always going to yield better miss rates? If not, give a counter example using an address stream.

Problem 2.2: Pipelined Cache Access

This problem requires the knowledge of Lecture 6 and 7. Please, read these materials before answering the following questions. You may also want to take a look at pipeline lectures (Lecture 4 and 5) if you do not feel comfortable with the topic.

Problem 2.2.A

Ben Bitdiddle is designing a five-stage pipelined MIPS processor with separate 32 KB direct-mapped primary instruction and data caches. He runs simulations on his preliminary design, and he discovers that a cache access is on the critical path in his machine. After remembering that pipelining his processor helped to improve the machine's performance, he decides to try applying the same idea to caches. Ben breaks each cache access into three stages in order to reduce his cycle time. In the first stage the address is decoded. In the second stage the tag and data memory arrays are accessed; for cache reads, the data is available by the end of this stage. However, the tag still has to be checked—this is done in the third stage.

After pipelining the instruction and data caches, Ben's datapath design looks as follows:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check	Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-back

Alyssa P. Hacker examines Ben's design and points out that the third and fourth stages can be combined, so that the instruction cache tag check occurs in parallel with instruction decoding and register file read access**. If Ben implements her suggestion, what must the processor do in the event of an instruction cache tag mismatch? Can Ben do the same thing with load instructions by combining the data cache tag check stage with the write-back stage? Why or why not?**

Problem 2.2.B

Alyssa also notes that Ben's current design is flawed, as using three stages for a data cache access won't allow writes to memory to be handled correctly. She argues that Ben either needs to add a fourth stage or figure out another way to handle writes**. What problem would be encountered on a data write? What can Ben do to keep a three-stage pipeline for the data cache?**

Problem 2.2.C

With help from Alyssa, Ben streamlines his design to consist of eight stages (the handling of data writes is not shown):

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back

Both the instruction and data caches are still direct-mapped. **Would this scheme still work with a set-associative instruction cache? Why or why not? Would it work with a set-associative data cache? Why or why not?**

Problem 2.2.D

After running additional simulations, Ben realizes that pipelining the caches was not entirely beneficial, as now the cache access latency has increased. **If conditional branch instructions resolve in the Execute stage, how many cycles is the processor's branch delay?**

Problem 2.2.E

Assume that Ben's datapath is fully-bypassed. When a load is executed, the data becomes available at the end of the D-cache Array Access stage. However, the tag has not yet been checked, so it is unknown whether the data is correct. If the load data is bypassed immediately, before the tag check occurs, then the instruction that depends on the load may execute with incorrect data**. How can an interlock in the Instruction Decode stage solve this problem? How many cycles is the load delay using this scheme (assuming a cache hit)?**

Problem 2.2.F

Alyssa proposes an alternative to using an interlock. She tells Ben to allow the load data to be bypassed from the end of the D-Cache Array Access stage, so that the dependent instruction can execute while the tag check is being performed. If there is a tag mismatch, the processor will wait for the correct data to be brought into the cache; then it will re-execute the load and all of the instructions behind it in the pipeline before continuing with the rest of the program. **What processor state needs to be saved in order to implement this scheme? What additional steps need to be taken in the pipeline? Assume that a DataReady signal is available and isasserted when the load data is available in the cache, and is set to 0 when the processor restarts its execution (you don't have to worry about the control logic details of this signal). How many cycles is the load delay using this scheme (assuming a cache hit)?**

Problem 2.3: Loop Ordering

This problem requires knowledge of Lecture 7. Please, read it before answering the following questions.

This problem evaluates the cache performances for different loop orderings. You are asked to consider the following two loops, written in C, which calculate the sum of the entries in a 128 by 64 matrix of 32-bit integers:

Loop A	***Loop B***

`sum = 0;`
`for (i = 0; i < 128; i++)`
`  for (j = 0; j < 64; j++)`
`  sum += A[i][j];`
	`sum = 0;`
	`for (j = 0; j < 64; j++)`
	`  for (i = 0; i < 128; i++)`
	`  sum += A[i][j];`

The matrix `A` is stored contiguously in memory in row-major order. Row major order means that elements in the same row of the matrix are adjacent in memory as shown in the following memory layout:

`A[i][j]` resides in memory location `[4*(64*i + j)]`

Memory Location:

0	4		252	256		4*(64*127+63)
A[0][0]	A[0][1]	...	A[0][63]	A[1][0]	...	A[127][63]

For *Problem 2.3.A* to *Problem 2.3.C*, assume that the caches are initially empty. Also, assume that only accesses to matrix `A` cause memory references and all other necessary variables are stored in registers. Instructions are in a separate instruction cache.

Problem 2.3.A

Consider a 4KB direct-mapped data cache with 8-word (32-byte) cache lines.
Calculate the number of cache misses that will occur when running Loop A.
Calculate the number of cache misses that will occur when running Loop B.

The number of cache misses for Loop A:_____

The number of cache misses for Loop B:_____

Problem 2.3.B

Consider a direct-mapped data cache with 8-word (32-byte) cache lines. Calculate the minimum number of cache lines required for the data cache if Loop A is to run without any cache misses other than compulsory misses. Calculate the minimum number of cache lines required for the data cache if Loop B is to run without any cache misses other than compulsory misses.

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

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

Problem 2.3.C

Consider a 4KB fully-associative data cache with 8-word (32-byte) cache lines. This data cache uses a first-in/first-out (FIFO) replacement policy.
Calculate the number of cache misses that will occur when running Loop A.
Calculate the number of cache misses that will occur when running Loop B.

The number of cache misses for Loop A:_____

The number of cache misses for Loop B:_____

Problem 2.4: Microtagged Cache

In this problem, we explore *microtagging*, a technique to reduce the access time of set-associative caches. Recall that for associative caches, the tag check must be completed before load results are returned to the CPU, because the result of the tag check determines which cache way is selected. Consequently, the tag check is often on the critical path.

The time to perform the tag check (and, thus, way selection) is determined in large part by the size of the tag. We can speed up way selection by checking only a subset of the tag—called a *microtag*—and using the results of this comparison to select the appropriate cache way. Of course, the full tag check must also occur to determine if the cache access is a hit or a miss, but this comparison proceeds in parallel with way selection. We store the full tags separately from the microtag array.

We will consider the impact of microtagging on a 4-way set-associative 16KB data cache with 32-byte lines. Addresses are 32 bits long. Microtags are 8 bits long. The baseline cache (i.e. without microtagging) is depicted in Figure H2-B in the attached handout. Figure 1, below, shows the modified tag comparison and driver hardware in the microtagged cache.

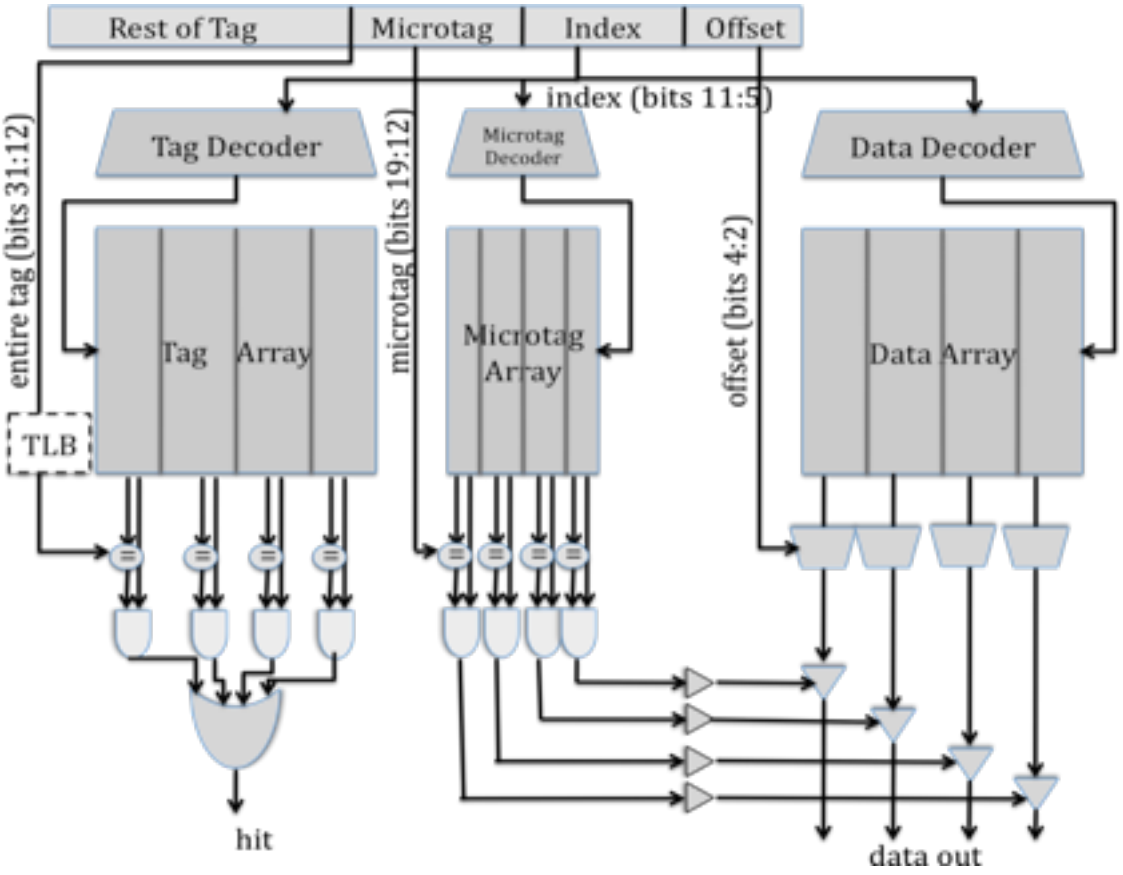

Figure 2.4-1: Microtagged cache datapath

Table 2.4-1, below, contains the delays of the components within the 4-way set-associative cache, for both the baseline and the microtagged cache. For both configurations, determine the critical path and the cache access time (i.e., the delay through the critical path).

Assume that the 2-input AND gates have a 50ps delay and the 4-input OR gate has a 100ps delay.

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

Table 2.4-1: Delay of each cache component

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

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

Problem 2.4.B **Microtagged Cache: AMAT**

Assume temporarily that both the baseline cache and the microtagged cache have the same hit rate, 95%, and the same average miss penalty, 20 ns. Using the cycle times computed in 2.4.A as the hit times, compute the average memory access time for both caches. **What was the old AMAT (in ns)? What is the new AMAT (in ns)?**

Problem 2.4.C **Microtagged Cache: Constraints**

Microtags add an additional constraint to the cache: in a given cache set, all microtags must be unique. This constraint is necessary to avoid multiple microtag matches in the same set, which would prevent the cache from selecting the correct way.

State which of the 3C's of cache misses this constraint affects. **How will the cache miss rate compare to an ordinary 4-way set-associative cache? How will it compare to that of a direct-mapped cache of the same size?**

Problem 2.4.D **Virtual Memory: Aliasing**

We now consider the effects of address translation on the microtagged cache. To avoid placing the TLB on the critical path, we use virtual microtags (i.e., the microtags are a subset of the virtual tag). The complete tags, however, are physical.

Without further modifications, it is possible for aliasing to occur in this cache? Carefully explain why this is the case.

Problem 2.4.E **Virtual Memory: Anti-Aliasing**

Propose an efficient solution to prevent aliasing in this cache. (Hint: it is not necessary to use the technique discussed in lecture, i.e. using the L2 cache to detect aliases.)

Problem 2.5: 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** to receive full credit.

	Compulsory Misses	Conflict Misses	Capacity Misses
Double the associativity (capacity and line size constant)			
Halving the line size (associativity and # sets constant)			
Doubling the number of sets (capacity and line size constant)			
Adding prefetching			

Problem 2.6: 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** to receive full credit.

	Hit Time	Miss Rate	Miss Penalty
Double the associativity (capacity and line size constant)			
Halving the line size (associativity and # sets constant)			
Doubling the number of sets (capacity and line size constant)			
Adding prefetching			