CS152/252A
Fall 2026

Caches and the Memory Hierarchy
Problem Set #2

Due: September 27 @ 11:59:59 PM PST

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 their own solution to the problems.

The problem sets also provide essential background material for the exam and the midterms. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets yourself you are unlikely to succeed on the exam or midterms!

By grading primarily on an effort basis, we mean that we will award significant partial credit for demonstrating your understanding of the problem and concepts at hand. As long as reasonable assumptions and explanations are provided, we will lean towards awarding credit.

We will distribute solutions to the problem set after the deadline to give you feedback.

Assignments must be submitted through Gradescope by **11:59:59pm PT** on the specified due date. *Box/clearly mark all solutions that don't involve filling in a figure/table. Only boxed/clearly marked solutions and filled in figures/tables will be considered for grading.* See the course website for the policy on late submissions.

Name: SOLUTIONS

SID:

Collaborators (Name, SID):

PROBLEMS 1, 2, and 3, and ARE ONLINE-ONLY

Submit all answers directly in the Gradescope free-response fields.

Do not include answers to Problems 1, 2, or 3 in your uploaded PDF.

Problem 1: Software Prefetching

Suppose we add software prefetching. Assume that each iteration of the inner loop takes 20 cycles (when there is a L1 miss). Of those 20 cycles, 15 cycles is the L1 miss penalty while 5 cycles is taken by the addition operation. Ignore any overheads from executing other instructions, including the software prefetch instruction. The prefetcher only prefetches when *prefetch_cond* becomes true (which is set by instructions that are not shown, and that you likewise do not need to model).

```
long sum = 0;
long matrix[64][64];
for (int i = 0; i < 64; i++) {
  for (int j = 0; j < 64; j++) {
    if (prefetch_cond) prefetch((char *)&matrix[i][j] + OFFSET);
    sum += matrix[i][j];
  }
}
```

Problem 1.1

Assume the cache has 64 byte cache lines and that each `long` is 8 bytes. What is the minimum OFFSET, in bytes, needed to fully hide the L1 miss penalty?

Each inner-loop iteration takes 20 cycles when there is a miss. Of these, 15 cycles correspond to the miss penalty, while 5 cycles are spent performing the addition. To completely hide the miss latency, the data must be prefetched early enough so that the 15-cycle miss penalty overlaps with useful computation.

Since each iteration performs 5 cycles of useful work, the prefetch must be issued:

15 / 5 = 3 iterations, before the data is needed.

Each element of matrix is a long, which occupies 8 bytes. Therefore, advancing by 3 iterations corresponds to:

3×8=24 bytes ahead of the current access.

Thus, OFFSET = 24.

Problem 1.2

Give a *prefetch_cond* that issues exactly one useful prefetch for each upcoming cache line and minimizes the total execution time.

Each L1 miss incurs a 15-cycle penalty, and each iteration performs 5 cycles of useful work. Therefore, the prefetch must be issued 15/5=3 iterations before the data is needed.

A 64-byte cache line holds 8 long elements, so a cache miss occurs every 8 iterations (when j is a multiple of 8). To hide the miss latency, the prefetch should be issued 3 iterations before the next cache line is accessed.

Therefore, the prefetch condition is: (j + 3) % 8 == 0, which triggers the prefetch three iterations before the next cache-line access.

Problem 1.3

Assume the matrix elements are stored in row-major order, how many accesses that previously missed will now hit in the cache?

In row-major order, the inner loop accesses memory sequentially. A 64-byte cache line holds 8 long elements, so each inner loop of 64 elements incurs: 64/8=8 cache misses without prefetching.

Across 64 rows, this results in: 64×8=512 cache misses.

If we assume all the rows are stored contiguously in the memory, then with software prefetching, the next cache line is fetched early, so its miss latency is hidden. Only the very first inner loop (i==0, j==0) cannot be prefetched and still cause a miss. Therefore, there is only 1 miss after applying prefetching.

Thus, software prefetching reduces the number of cache misses by: 512 − 1 = 511

Problem 1.4

Why is software prefetching hard to make effective in practice? If we want to use hardware prefetcher instead, what kind of hardware prefetcher works well for this access pattern?

Choosing the correct prefetch distance (timing) is very challenging in practice.

If we want to switch to hardware prefetcher, a stride prefetcher will be ideal.

Problem 2: 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.**

For subparts where the outcome is ambiguous, pick one outcome and answer with reasonable assumptions and explanations.

Each row corresponds to one Gradescope free-response part.

		Compulsory Misses	**Conflict Misses**	**Capacity Misses**
2.1)	Doubling the size of a matrix that needs to be traversed. The matrix is traversed multiple times. Matrix lines are evenly distributed across sets	**Increase.** Larger matrix means more lines needs to be loaded in for the first time.	**No effect.** The sets are evenly distributed and having an infinetly large cache would result in zero conflict misses	**Increase.** Increasing capacity by infinity would result in only compulsory misses
2.2)	Changing the matrix from Problem 1 to be from row major to column major order. The matrix is larger than the size of the cache.	**No effect** the same total amount of data is being loaded into the cache for the first time	**Increase.** The same row will be loaded in, booted out, and loaded in again	**No effect / Increase.** Depends on if the amount of rows in the cache equals the amount of rows in the matrix
2.3)	Double cache capacity and associativity; line size stays constant	**No effect.**	**Decrease** Higher associativity gives blocks more possible locations to choose from without conflicting	**Decrease.** Greater capacity.

Problem 3: Memory Hierarchy Performance

Problem 3.1 Cache Design Tradeoffs

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**.

For subparts where the outcome is ambiguous, pick one outcome and answer with reasonable assumptions and explanations.

Each row corresponds to one Gradescope free-response part.

		Hit Time	Miss Rate	Miss Penalty
a)	Halving the line size (associativity and number of sets constant)	**Decreases.** The cache is now physically smaller, which overshadows the slightly increased tag-check time (the tag grows by 1 bit).	**Increases.** Smaller capacity and less ability to take advantage of spatial locality within a single cache line (more compulsory misses).	**Decreases.** Smaller lines can be brought in more quickly. **Or: No effect,** because the cache already brings in the critical word first.
b)	Doubling the number of sets (capacity and line size constant)	**Decreases.** The number of sets increases, so tags get smaller. Fewer tags must be checked, and fewer ways have to be muxed out.	**Increases.** More conflict misses because associativity gets halved.	**No effect.** This is dominated by the outer memory hierarchy.
c)	Combine L1 ICache and DCache into one cache with the combined capacity (associativity and line size constant)	**Increases.** If the cache is dual-ported, it will be slower than a single-ported cache. If there is a single port, data accesses may frequently stall for instruction accesses, or vice versa.	**May decrease.** The cache can more flexibly allocate space to either data or instructions depending on dynamic program behavior. **May increase** in edge cases due to conflicts between instruction and data accesses.	**No effect.** This is dominated by the outer memory hierarchy.

Problem 3.2 World Class CPU

Imagine you're tweaking your world class CPU from HW1, and you want to optimize your cache. Provide three unique (as in not from problem 3.1) optimizations you would make to your cache, and explain how/when it benefits cache performance. Feel free to be creative! Just provide a justification for your choice.

Each row corresponds to one Gradescope free-response part.

	Optimization	**Impact**
a)	Critical word first	Reduces the miss penalty for addresses not at the start of a cache line.
b)	Non-blocking cache	Supporting miss-under-miss reduces the effective miss penalty.
c)	Prefetching	A good prefetcher reduces compulsory misses.

Submit the following as either
a typed PDF
or a scanned document.

Problem 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 Handout #2. Figure 1, below, shows the modified tag comparison and driver hardware in the microtagged cache.

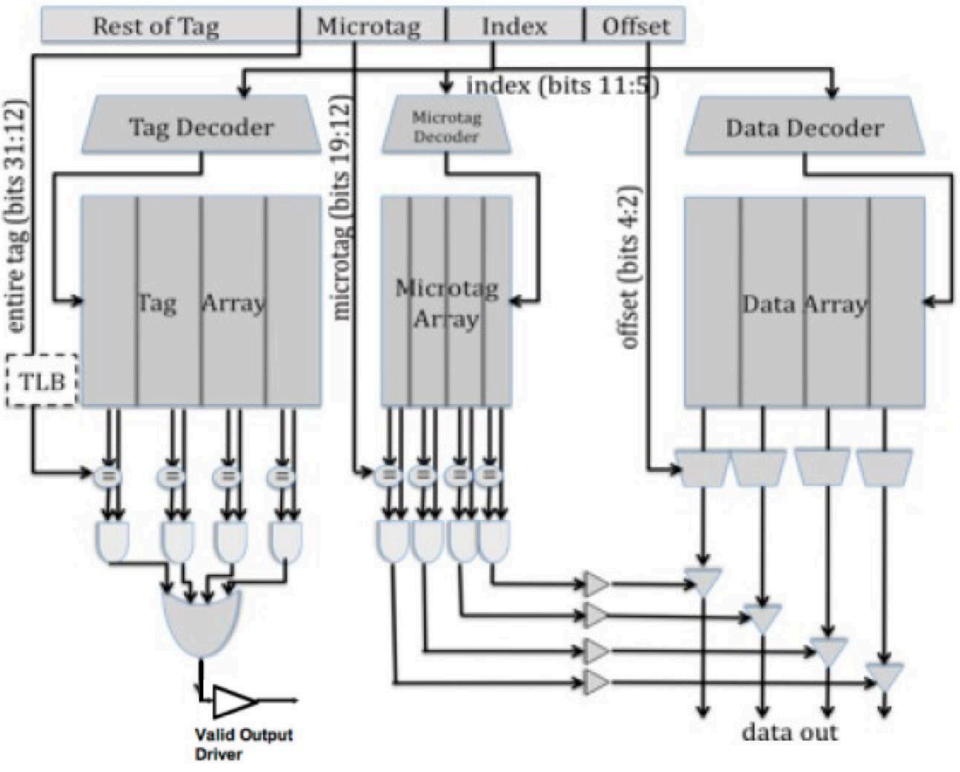


Figure 2.4-1: Microtagged cache datapath

Problem 4.1 Cache Cycle Time

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
		Microtag		240
Memory array	$20 \cdot \log_2$ (# of rows) + $20 \cdot \log_2 \lceil$(# of bits in a row)$\rceil$ + 100	Tag	380	380
		Data	440	440
		Microtag		340
Comparator	20·(# of tag bits) + 100	Tag	500	500
		Microtag		260
N-to-1 MUX	$50 \cdot \log_2$ 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

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

Candidate 1: Full tag check

tag decoder ⟶ tag read ⟶ comparator ⟶ 2-in AND ⟶ 4-in OR ⟶ valid output driver

240 ps + 380 ps + 500 ps + 50 ps + 100 ps + 100 ps = 1370 ps

Candidate 2: Data select based on full tag check

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

240 ps + 380 ps + 500 ps + 50 ps + 200 ps + 300 ps = 1670 ps

Candidate 3: Data readout

data decoder ⟶ data read ⟶ 4-to-1 MUX ⟶ data output driver

240 ps + 440 ps + 250 ps + 300 ps = 1230ps

The critical path is the data select based on the full tag match. The cycle time is 1670 ps.

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

Candidate 1: Full tag check

same as baseline full tag check => 1370 ps

Candidate 2: Data select based on microtag check

μtag decoder ⟶ μtag read ⟶ comparator ⟶ 2-in AND ⟶ buffer driver ⟶ data out driver

240 ps + 340 ps + 260 ps + 50 ps + 200 ps + 300 ps = 1390 ps

Candidate 3: Data readout

same as baseline data read => 1230 ps

The critical path is the data select based on the microtag check. The cycle time is 1390 ps.

Problem 4.2 AMAT

Assume temporarily that both the baseline cache and the microtagged cache have the same hit rate, 90%, and the same average miss penalty, 15 ns. Using the hit times of 1.5 ns and 1.2 ns for the baseline and microtag caches respectively, compute the average memory access time for both caches.

i) What was the old baseline AMAT (in ns)?

ii) What is the new AMAT (in ns)?

AMAT = (hit_time) + (miss)_rate x (miss_penalty)

= X + (0.1) * (15ns) = X + 1.5ns, where X is the hit time

Old AMAT = 1.5 + 1.5 = 3 ns

New AMAT with microtags = 1.2 + 1.5 = 2.7 ns

Problem 4.3 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.

i) State which of the 3C's of cache misses this constraint affects.

ii) How will the cache miss rate compare to an ordinary 4-way set-associative cache?

iii) How will it compare to that of a direct-mapped cache of the same size?

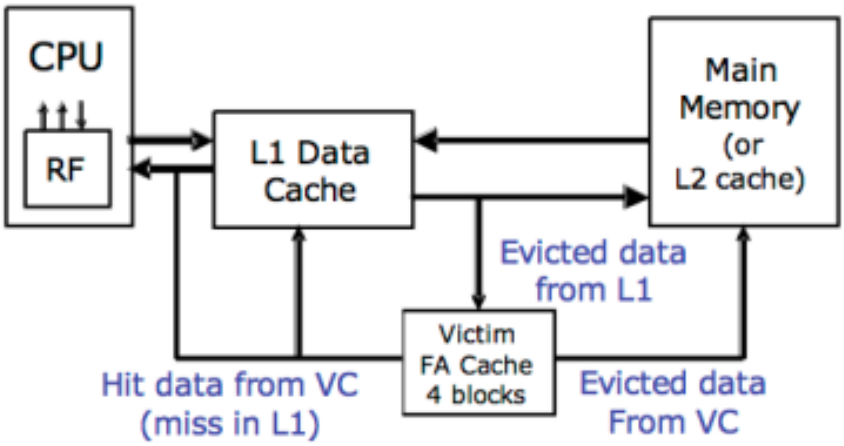
iv) Explain why bits 19 through 12 might be a good choice for the microtag, or propose a different choice.

Because the uniqueness property of microtags restricts the replacement policy, the cache isn't free to make as optimal replacement decisions as it could in the baseline. This will lead to some increase in conflict misses. The magnitude of this effect depends on which 8 bits are selected to form the microtag. In principle, using the bottom 8 bits would result in more potential for microtag collisions and would add the biggest restriction to the ability of the cache to hold spatially local data – data within 2^{12} to 2^{20} bytes of each other. The same argument could be used for choosing the top 8 bits. When addressing data that is spatially local, it will likely have the same upper tag bits. But due to our constraint of uniqueness, this would also cause many cache conflicts. Thus, we may want to choose 8 bits somewhere in the middle of the tag, depending on our application. Regardless, the microtagged cache will still be better than a direct mapped cache of the same size and line size.

Problem 5: Victim Cache Evaluation

Although direct-mapped caches have an advantage of smaller access time than set- associative caches, they have more conflict misses due to their lack of associativity. In order to reduce these conflict misses, Norm Jouppi proposed victim caching, where a small fully-associative back up cache, called a victim cache, is added to a direct-mapped L1 cache to hold recently evicted cache lines.

The following diagram shows how a victim cache can be added to a direct-mapped L1 data cache. Upon a data access, the following chain of events takes place:



(i) The L1 data cache is checked. If it holds the data requested, the data is returned.

(ii) If the data is not in the L1 cache, the victim cache is checked. If it holds the data requested, the data is moved into the L1 cache and sent back to the processor. The data evicted from the L1 cache is put in the victim cache, and put at the end of the FIFO replacement queue.

(iii) If neither of the caches holds the data, it is retrieved from memory, and put in the L1 cache. If the L1 cache needs to evict old data to make space for the new data, the old data is put in the victim cache and placed at the end of the FIFO replacement queue. Any data that needs to be evicted from the victim cache to make space is written back to memory or discarded, if unmodified.

Note that the two caches are *exclusive*. That means that the same data cannot be stored in both L1 and victim caches at the same time.

Problem 5.1 Baseline Cache Design (Optional)

The diagram below shows our victim cache, a 32-byte fully associative cache with four 8-byte cache lines. Each line contains two 4-byte words and has an associated tag and two status bits (valid and dirty). The Input Address is 32-bits. Since the cache is word-addressed, it does not use the two least significant bits. The output of the cache is a 4-byte word.

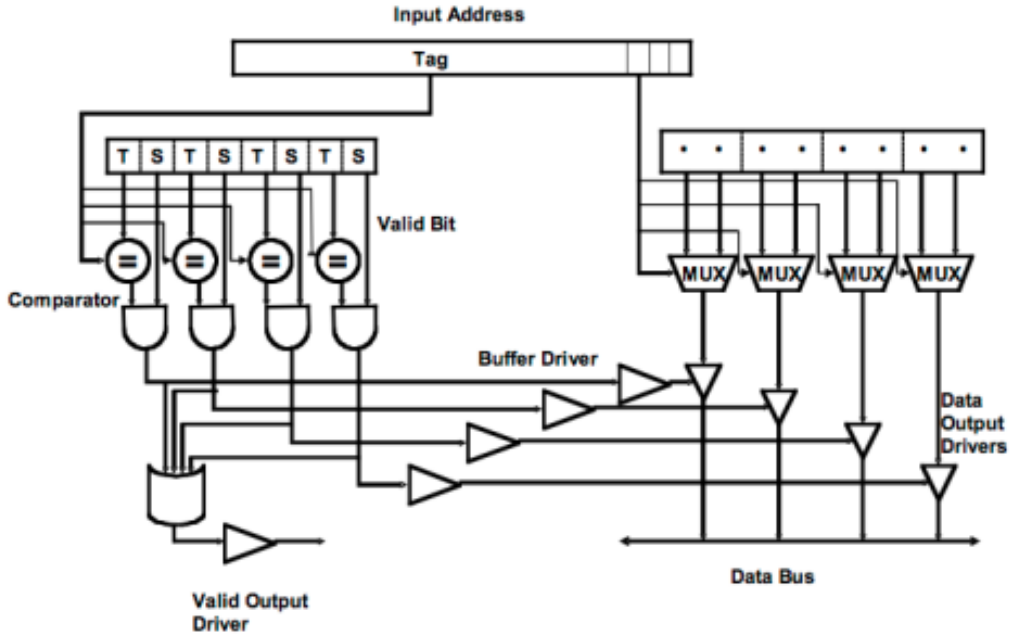


Figure 2.5-1: Victim cache datapath

Please complete Table 2.5-1 with delays across each element of the cache. Using the data you compute in Table 2.5-1, calculate the critical path delay through this cache (from when the Input Address is set to when both Valid Output Driver and the appropriate Data Output Driver are outputting valid data).

Component	Delay equation (ps)	FA(ps)
Comparator	30·(# of tag bits) + 100	970
N-to-1 MUX	50·$\log_2$ N + 100	150
Buffer driver	200	200
AND gate	100	100
OR gate	50· $\log_2$ N + 100	200
Data output driver	50·(associativity) + 100	300
Valid output driver	100	100

Table 2.5-1: Delay of each cache component

Below, we evaluate the three major paths through the victim cache to find the critical path and cycle time. Note that the victim cache is fully-associative and uses 29-bit tags.

Candidate 1: Tag check

comparator ⟶ 2-in AND ⟶ 4-in OR ⟶ valid output driver

970 ps + 100 ps + 200 ps + 100 ps = 1370 ps

Candidate 2: Data select based on tag check

comparator ⟶ 2-in AND ⟶ buffer driver ⟶ data output driver

970 ps + 100 ps + 200 ps + 300 ps = 1570 ps

Candidate 3: Data readout

2-to-1 MUX ⟶ data output driver

200 ps + 300 ps = 500 ps

The critical path is the data select based on the tag match. The cycle time is 1570 ps.

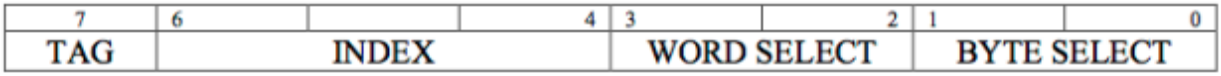
Problem 5.2 Victim Cache Behavior (Optional)

Now we will study the impact of a victim cache on cache hit rate.

Our main L1 cache is a 128 byte, direct-mapped cache with 16 bytes per cache line. The cache is word (4-bytes) addressable.

The victim cache is similar to the one in Figure 2.5-1. It is a 32-byte fully associative cache with 16 bytes per cache line and is also word addressable. (Note that these parameters are different from problem 5.1) It uses the first in first out (FIFO) replacement policy.

Please complete Table 2.5-2 showing a trace of memory accesses. In the table, each entry contains the tag of that line, or "inv", if no data is present. You should only fill in elements in the table when a value changes. For simplicity, the addresses are only 8 bits. The first 3 lines of the table have been filled in for you. For your convenience, the address breakdown for access to the main cache is depicted below.



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

Table 2.5-2: Memory access trace

Problem 5.3 Average Memory Access Time

Assume **15%** of L1 misses are resolved in the victim cache. Checking and retrieving data from the victim cache takes **4 cycles**. If the access also misses in the victim cache, retrieving the data from main memory takes an additional **50 cycles**. By how many cycles does the victim cache improve the average memory access time? Assume the L1 miss rate is **10%**.

AMAT = HitTime + L1MissRate * L1MissPenalty

AMAT2 = HitTime + L1MissRate * (VictimHitTime + (1 - VictimHitRate) * VictimMissPenalty)

VictimMissPenalty = L1MissPenalty = DRAMTime, since this is just time to get data from main memory

AMAT – AMAT2 = L1MissRate * (DRAMTime – VictimHitTime - (1 – VictimHitRate) * DRAMTime)

= 0.1 * (50 – 4 – 0.85 * 50)

= 0.1 * 3.5 = 0.35