# CS152 Computer Architecture and Engineering
# CS252 Graduate Computer Architecture
Spring 2019

# SOLUTIONS

Caches and the Memory Hierarchy

*Assigned February 13* Problem Set #2 *Due Wed, February 27*

**http://inst.eecs.berkeley.edu/~cs152/sp19**

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

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) + $20\times \log_2$ (# of bits in a row) + 100 | Tag | 422 | 425 |
| | | 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 has 32-byte lines, which means that there are 5 offset bits. However, we are also told that the bottom two bits are ignored, which appears in the calculation for the word-select mux delay, which is an 8-way mux.

In a 128 KB direct-mapped cache with 8 word (32B) cache lines, there are 4096 = $2^{12}$ cache lines (128KB/32B). Therefore, there are 12 index bits. Subtracting the 12 index bits and the 5 offset bits from the 32-bit address, it is clear that there must be 15 tag bits.

We also need the number of rows and the number of bits per row in the tag & data memories. The number of rows is simply the number of cache lines (4096), 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), which yields 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 for a 32-byte cache line.

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 × $\log_2$(# of rows) + 20 × $\log_2$(# bits in a row) + 100***
***= 20 × $\log_2(2^{12})$ + 20 × $\log_2(17)$ + 100 ≈ 422 ps***
***Memory array (Data) = 20 × $\log_2$(# of rows) + 20 × $\log_2$(# bits in a row) + 100***
***= 20 × $\log_2(2^{12})$ + 20 × $\log_2(256)$ + 100 = 500 ps***

***Comparator = 20 × (# of tag bits) + 100 = 20 × 15 + 100 = 400 ps***

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

***Data output driver = 50 × (associativity) + 100 = 500 × 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 (tag check and data read). By taking the maximum delay of these two paths, we are left with the critical path.

***Time to tag check valid driver***
***= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time) + (valid output driver time)***
***≈ 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***

From the above calculations, we see that the tag check is the critical path. ***The access time is 1312 ps. At 1.5 GHz, this cache access takes (1312 ps / (1 / 1.5GHz)) ≈ 2 cycles. Here, rounding up to the nearest cycle is sensible, as this reflects how a synchronous system would actually work.***

## Problem 1.B — Access Time: Set-Associative

As in 2.1.A, we have a 32-byte cache lines and a 5-bit offset, of which only the top three bits are used to select an output word. However, the number of tag and index bits has changed.

**Index bits:** ((128kB cache / 4 ways) / 32B line) = 1024 sets => **10 index bits**
**Tag bits:** (32-bit address) - (10-bit index) - (5-bit offset) => **17 tag bits**

**# of rows in the tag/data memories:** 1024 sets => **1024 rows**

**# of bits in a tag memory row:** 4 x (17-bit tag + 2-bit status) => **76 bits**
**# of bits in a data memory row:** 4 x (32B line) => **1024 bits**

We now have all the numbers required to fill in the table for the SA cache.

***Decoder (Tag) = 20 × (# of index bits) + 100 = 20 × 10 + 100 = 300 ps***
***Decoder (Data) = 20 × (# of index bits) + 100 = 20 × 10 + 100 = 300 ps***
***Memory array (Tag) = 20 × $log_2$(1024) + 20 × $log_2$(76) + 100 = 425 ps***
***Memory array (Data) = 20 × $log_2$(1024) + 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***
***Valid output driver = 100 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, so we must calculate the delay of each.

***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 is the path to select the appropriate data output driver based on the tag check. ***The access time is 1715 ps***. ***At 1.5 GHz, this cache access takes (1715 ps / (1 / 1.5GHz)) ≈  3 cycles. Here, rounding up to the nearest cycle is sensible, as this reflects how a synchronous system would actually work.***

For the 4-way set-associative cache, there are 8. As always, we start by computing the bit positions of the tag, index, and offset fields. **All addresses in the table are in HEXADECIMAL.**

```
address = 12 bits addr[11:0]
tag     =  5 bits addr[11:7]   (12 bits - 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. Remember, the tag, index, and offset fields of the address all change in width! **All addresses in the table are in HEXADECIMAL.**

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

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

| | **4-way LRU** |
|---|---|
| **Total Misses** | 8 |
| **Total Accesses** | 13 |

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

***For the direct-mapped cache, the average memory access latency would be:***
***(2 cycles) + (10/13) × (20 cycles) = 17.4 cycles.***

***For the LRU set-associative cache, the average memory access latency would be:***
***(3 cycles) + (8/13) × (20 cycles) = 15.3 cycles.***

***For the FIFO set-associative cache, the average memory access latency would be:***
***(3 cycles) + (9/13) × (20 cycles) = 16.8 cycles.***

The set-associative cache with LRU replacement is better than the direct-mapped cache 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. ***In this case, the LRU policy took better advantage of temporal locality***.

LRU does not always outperform FIFO. Assume we have a set-associative cache with the same parameters as in 1.C and an access sequence shown below. There is a miss with LRU for the last access while there is a hit with FIFO.

```
0x100
0x120
0x140
0x160
0x100
0x180
0x120
```

## Problem 2: Loop Ordering

### Problem 2.A

Each element of the 128x32 matrix A can only be mapped to *one* particular cache location in this direct-mapped data cache. Since each row has 32 32-bit integers, and since each cache line can hold 8 32-bit words, a row of the matrix occupies the lines in four consecutive sets of the cache.

Loop A—where each iteration of the inner loop sums a row of A—accesses memory addresses in a linear sequence. Given this access pattern, the access to the first word in each cache line will miss, but the next seven accesses will hit. After sequentially moving through this line, it will not be accessed again, so its later eviction will not cause any future misses. Therefore, Loop A will only have compulsory misses for the 512 (128 rows x 4 lines per row) that matrix A spans.

The consecutive accesses in Loop B will move in a stride of 32 bytes. Therefore, the inner loop will touch the first element in 128 cache lines before the next iteration of the outer loop. While intuition might suggest that the 128 lines could all fit in the cache with 128 sets, there is a complicating factor: each row is *four* cache lines past the previous row, meaning that the lines accessed when traversing the first column go in indices 0, 4, 8, 12, and so on. Since the lines containing the column are competing for only one fourth of the sets, the lines loaded when starting a column are evicted by the time the column is complete, preventing any reuse. Therefore, all 4096 (128 x 32) accesses miss.

The number of cache misses for Loop A:________512___________

The number of cache misses for Loop B:________4096__________

### Problem 2.B

Since *Loop* A accesses memory sequentially, we can sum all the elements in a cache line and then never touch it again. Therefore, we only need to hold 1 active line at any given time to avoid all but compulsory misses.

For Loop B to run without any cache misses other than compulsory misses, the data cache needs to have the ability to hold one column of matrix A in the cache. Since the consecutive accesses in the inner loop of Loop B will use every fourth cache line, and since we have 128 rows, Loop B requires 512 (128 × 4) lines to avoid all but compulsory misses.

Data-cache size required for Loop A: _________1_________ cache line(s)

Data-cache size required for Loop B: _______512_________ cache line(s)

## Problem 2.C

Loop A still only has 512 (128 rows x 4 lines per row) compulsory 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 fourth cache line. Therefore, we can fit 8 columns in the cache after taking one compulsory miss on each, and we can reuse all the remaining elements in each line before evicting them. This means that loop B experiences only (128 rows x 4 lines per row) compulsory misses.

The number of cache misses for Loop A:______512_________

The number of cache misses for Loop B:______512_________

## 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 | 369 | 369 |
| | | 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 |

**Note: we use a valid bit in the microtag array and two status bits in the tag array. Solutions based on other schemes may be accepted with sufficient justification.**

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 + 369 ps + 500 ps + 50 ps + 100 ps + 100 ps = 1359 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 + 369 ps + 500 ps + 50 ps + 200 ps + 300 ps = **1659 ps**

Candidate 3: Data readout
data decoder → data read → 4-to-1 MUX → data output driver
240 ps + 440 ps + 250 ps + 300 ps = 1230 ps

The critical path is the data select based on the full tag match. The cycle time is 1659 ps.

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

Candidate 1: Full tag check
same as baseline => 1359 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 => 1230 ps

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

## Problem 3.B AMAT

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

Old AMAT = 1.62 + 1 = 2.62 ns

New AMAT with microtags = 1.356 + 1 = 2.356 ns

## Problem 3.C Constraints

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. However, it will still be better than a direct-mapped cache of the same size and line size.

## Problem 4: Victim Cache Evaluation

### Problem 4.A Victim Cache Design

| Component | Delay equation (ps) | Delay (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 |

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
680 ps + 100 ps + 200 ps + 100 ps = 1080 ps

Candidate 2: Data select based on tag check
comparator → 2-in AND → buffer driver → data output driver
680 ps + 100 ps + 200 ps + 300 ps = **1280 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 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) | No effect<br><br>Doubling associativity doesn't change when lines are first brought into the cache | Decrease<br><br>Typically, higher associativity reduces conflict misses, since there are more places to put the same element. | No effect<br><br>Capacity does not change. |
| Halving the line size (associativity and # sets constant)<br><br>Halves capacity! | Increase<br><br>Shorter lines mean fewer adjacent elements are brought in with the first access to a given line. | Increase<br><br>The program will access more cache lines in total, creating more opportunity for conflict misses. | Increase<br><br>Capacity has been cut in half. |
| Doubling the number of sets (capacity and line size constant)<br><br>Halves associativity! | No effect<br><br>Halving associativity doesn't change when lines are first brought into the cache | Increase<br><br>Typically, lower associativity increases conflict misses, since there are fewer places to put the same element. | No effect<br><br>Capacity does not change. |

| | Compulsory Misses | Conflict Misses | Capacity Misses |
|---|---|---|---|
| Adding prefetching | Decrease<br><br>Ideally, a good prefetcher can bring data in before we use it, avoiding compulsory misses. | **Best answer: Decrease**<br>**With good prefetching, conflict misses should decrease, as the prefetcher will often bring lines that have been evicted back into the cache.**<br><br>Okay answer: increase<br>With mediocre prefetching, conflict misses could increase, as the prefetcher could evict useful lines to bring in useless.<br><br>Other okay answer: no effect<br>With mediocre prefetching that uses a stream buffer or other ancillary structure, there will be little to no effect on conflict misses. | **Best answer: Decrease**<br>**With good prefetching, capacity misses should decrease. In a situation where the working set simply won't fit, the prefetcher can dynamically bring lines in, "Just-In-Time," avoiding what would have been capacity misses.**<br><br>Okay answer: no effect<br>With a mediocre prefetcher that would increase conflict misses, it is unlikely that capacity misses would increase. If prefetcher traffic evicts useful data, newly created misses will almost certainly be conflict misses. |

## 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)<br><br>Halves # of sets | Increases<br><br># of sets decreases, so tags get larger. More tags must be checked, and more ways have to be muxed outs. | Decrease<br><br>Fewer conflict misses. | No effect<br><br>This is dominated by the outer memory hierarchy |
| Halving the line size (associativity and # sets constant)<br><br>Halves capacity | Decreases<br><br>The cache is now physically smaller, which overshadows the slightly increased tag check time (tag grows by 1 bit). | Increases<br><br>Smaller capacity, less ability to take advantage of spatial locality within a single cache line (more compulsory misses). | Decreases<br><br>Smaller lines can be brought in more quickly.<br><br>**OR**<br><br>No effect, because cache already brings in critical word first. |

| | | | |
|---|---|---|---|
| Doubling the number of sets (capacity and line size constant)<br><br>Halves associativity | Decreases<br><br># of sets increases, so tags get smaller. Fewer tags must be checked, and fewer ways have to be muxed outs. | Increases<br><br>More conflict misses because associativity gets halved. | No effect<br><br>This is dominated by the outer memory hierarchy |
| Adding prefetching | No effect<br><br>The prefetcher isn't on the hit path. | Decreases<br><br>The whole purpose of a prefetcher is to reduce the miss rate by bringing in data ahead of time. | Good answer: no effect.<br><br>May increase due to bandwidth pollution but we can(should) give a priority on cache misses over prefetch requests.<br><br>May decrease because a prefetch can be inflight when a miss occurs (but this is unlikely). |