Digital Design and Computer Architecture(252-0028-00L), Spring 2023
Optional HW 7: Memory Hierarchy, Caches, Prefetching, and Virtual Memory
SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Juan Gomez Luna, Mohammad Sadrosadati, Mohammed Alser, Ataberk Olgun, Giray Yaglikci, Can Firtina, Geraldo De Oliveira Junior, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Nisa Bostancı, Rakesh Nadig, Joel Lindegger, İsmail Emir Yüksel, Haocong Luo, Yahya Can Tuğrul, Julien Eudine

Released: Friday, June 23, 2023

1 Instruction and Data Caches

Consider the following loop is executed on a system with a small instruction cache (I-cache) of size 16 B. The data cache (D-cache) is fully associative of size 1 KB. Both caches use 16-byte blocks. The instruction length and data word size are 4 B. The initial value of register `$1` is 40. The value of `$0` is 0. (Note: Assume that the first instruction of the loop is aligned to the beginning of a cache block).

```
Loop: lw   $6, X($1)
      addi $6, $6, 1
      sw   $6, Y($1)
      subi $1, $1, 4
      beq $1, $0, Exit
      j    Loop
Exit:  ...
```

(a) Compute I-cache and D-cache miss rates, considering:

- `X` and `Y` are different arrays.
- `X` and `Y` are the same array.

First, we analyze a case with an infinite number of iterations (or a multiple of 4 iterations):
The I-cache can keep 4 instructions. Thus, in each iteration there will be 2 cache misses. The I-cache miss rate will be 2/6 = 0.33.

- `X` and `Y` are different arrays.
  If `X` and `Y` are different arrays, there will be 2 cache misses every 4 iterations, that is, one read miss and one write miss every 8 accesses. In that case, the D-cache miss rate will be 2/8 = 0.25.
- `X` and `Y` are the same array.
  If `X` and `Y` are the same array, the D-cache miss rate will be one half of the previous one (0.125).

Second, we consider 10 iterations of the loop:
In the last iteration, the jump is not executed. Thus, the I-cache miss rate is 20/59 = 0.34.

- `X` and `Y` are different arrays.
  If `X` and `Y` are different arrays, there will be 2 cache misses every 4 iterations, but for the last two iterations there will be 2 cache misses too. The D-cache miss rate will be 3/10 = 0.30.
- `X` and `Y` are the same array.
  If `X` and `Y` are the same array, the D-cache miss rate will be one half of the previous one (0.15).

(b) Compute the average number of cycles per instruction (CPI), using a baseline ideal CPI (ideal caches) equal to 2, and a miss latency equal to 10 clock cycles.

Since load and store instructions are one third of all the execute instructions, CPI is calculated as (for 10 iterations):
CPI = $2 + 0.34\times10 + 0.33\times0.30\times10 = 6.390$ cycles, if `X` and `Y` are different arrays.
CPI = $2 + 0.34\times10 + 0.33\times0.15\times10 = 5.895$ cycles, if `X` and `Y` are the same array.

(c) A compiler could unroll this loop for optimization. How would this affect CPI?

If the compiler unrolls the loop, the total number of executed instructions is reduced. As there will be one I-cache miss every four executed instructions, the I-cache miss rate will be $1/4 = 0.25$. The D-cache miss rate remains the same, but the fraction of loads and stores changes. `beq` and `j` instructions are no longer necessary, so the fraction of load and stores is 0.50.
The new CPI is:
CPI = $2 + 0.25\times10 + 0.50\times0.30\times10 = 6.00$ cycles, if `X` and `Y` are different arrays.
CPI = $2 + 0.25\times10 + 0.50\times0.15\times10 = 5.25$ cycles, if `X` and `Y` are the same array.

(d) How would the result of part (a) change with a 32-byte I-cache?

If the size of the I-cache is 32 B, the entire loop fits in it. There will be only two cold misses in the first iteration. Thus, the I-cache miss rate will be $2/59 = 0.033$.

2 Reverse Engineering Caches I

You're trying to reverse-engineer the characteristics of a cache in a system so that you can design a more efficient, machine-specific implementation of an algorithm you're working on. To do so, you've come up with four patterns that access various *bytes* in the system in an attempt to determine the following four cache characteristics:

- Cache block size (8, 16, 32, 64, or 128 B)
- Cache associativity (2-, 4-, or 8-way)
- Cache size (4 or 8 KB)
- Cache replacement policy (LRU or FIFO)

However, the only statistic that you can collect on this system is cache hit rate after performing the access pattern. Here is what you observe:

Access Pattern	Addresses Accessed (Oldest → Youngest)									Hit Rate
A	0	4096	8192	12288	16384	4096	0			1/7
B	0	1024	2048	3072	4096	5120	6144	3072	0	1/9
C	0	4	8	16	32	64	128	256	512	4/9
D	128	1152	2176	3200	128	4224	1152			2/7

Based on what you observe, what are the following characteristics of the cache? (Be sure to justify clearly your answer for full credit.)

(a) Cache block size (8, 16, 32, 64, or 128 B)?

Let's focus on access pattern C for this part. In this access pattern, for a given cache block size, all bytes map to certain sets—regardless of cache associativity and regardless of cache size.
There is only one mapping that causes 4 accesses out of 9 to be hits: At a cache block size of 64 B, addresses 4, 8, 16, and 32 all hit in the block brought in by the access to address 0. The other accesses go to different cache blocks.

(b) Cache associativity (2-, 4-, or 8-way)?

Let's focus on access pattern A for this part. In this access pattern, for a given cache associativity, all bytes map to the *same* set—regardless of cache block size and regardless of cache size.
There is only one associativity that causes 1 access out of 7 to be a hit: An associativity of 4 allows address 4096 to be a hit. Any less, and no accesses would hit; any more, and address 0 would also hit.

(c) Cache size (4 or 8 KB)?

Let's focus on access pattern B for this part. In this access pattern, given a cache associativity of 4, there is only one cache size for which all blocks map to the same set (regardless of cache block size), thus causing 1 access out of 9 to be a hit: A cache size of 4 KB causes all of the blocks to map to the same set, generating a hit for address 3072. At 8 KB, the access to address 0 also hits in the cache.

(d) Cache replacement policy (LRU or FIFO)?

Let’s focus on access pattern D for this part. In this access pattern, given a cache associativity of 4, and given a cache size of 4 KB, for a given replacement policy, all bytes map to the same set—regardless of cache block size.
First, notice that the access to address 128 will be a hit: Under LRU, the next block to be evicted would be the one for address 1152; whereas, under FIFO, the next block to be evicted would be the one for address 128.
Second, notice that the access to address 4224 then evicts the respective block. The access for address 1152 would then miss under LRU, but hit under FIFO, for a total of two hits (including the access to address 128). FIFO is the only cache replacement policy that causes 2 accesses out of 7 to be hits.

3 Tracing the Cache

Assume you have three toy CPUs: 6808-D, 6808-T, and 6808-F. All three CPUs feature one level of cache. The cache size is 128 bytes, the cache block size is 32 bytes, and the cache uses LRU replacement. The only difference between the three CPUs is the associativity of the cache:

- 6808-D uses a direct mapped cache.
- 6808-T uses a two-way associative cache.
- 6808-F uses a fully associative cache.

You run the SPECMem3000 program to evaluate the CPUs. This benchmark program tests only memory read performance by issuing read requests to the cache. Assume that the cache is empty before you run the benchmark. The cache accesses generated by the program are as follows, in order of access from left to right:

`A, B, A, H, B, G, H, H, A, E, H, D, H, G, C, C, G, C, A, B, H, D, E, C, C, B, A, D, E, F`

Each letter represents a unique cache block. **All 8 cache blocks are contiguous in memory.** However, the ordering of the letters does not necessarily correspond to the ordering of the cache blocks in memory. For 6808-D, you observe the following cache misses in order of generation:

`A, B, A, H, B, G, A, E, D, H, C, G, C, B, D, A, F`

(a) By using the above trace, please identify which cache blocks are in the same set for the 6808-D processor. Please be clear.

```
A and B
C and G
H and D
E and F
```

(b) Please write down the sequence of cache misses for the 6808-F processor in their order of generation. (Hint: You might want to write down the cache state after each request).

By simulating the cache and using the requests

```
Req:  A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F
M? :  x x   x   x       x   x   x x       x x x x x x   x x x x x
MRU:  A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F
   :  - A B A H B G G H A E H D H G G C G C A B H D E E C B A D E
   :  - - - B A H B B G H A E E D H H H H G C A B H D D E C B A D
LRU:  - - - - - A A A B G G A A E D D D D H G C A B H H D E C B A
```

(c) For 6808-T, you observed the following five cache misses in order of generation:

```
A, B, H, G, E
```

But, unfortunately, your evaluation setup broke before you could observe all cache misses for the 6808-T. Using the given information, which cache blocks are in the same set for the 6808-T processor?

We first simulate the cache up to the point it breaks. By simulating the cache and using the requests

```
Req :  A B A H B G H H A E
M?  :  x x   x   x       x
MRU0:  A B A A B B B B A E
LRU0:  - A B B A A A A B A
MRU1:  - - - H H G H H H H
LRU1:  - - - - - H G G G G
```

If H was in the same set as A and B, then the B right after H would have missed. Similarly if G was in the same set as A and B, the A right before E would have missed. Using this information and the sets calculated in part (a), the sets are respectively

```
A, B, E, and F
H, G, C, and D
```

(d) Please write down the sequence of cache misses for the 6808-T processor in their order of generation.

```
Req :  A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F
M?  :  x x   x   x       x   x   x x         x x x x x     x   x x
MRU0:  A B A A B B B B A E E E E E E E E E A B B B E E E B A A E F
LRU0:  - A B B A A A A B A A A A A A A A A E A A A B B B E B B A E
MRU1:  - - - H H G H H H H H D H G C C G C C C H D D C C C C D D D
LRU1:  - - - - - H G G G G G H D H G G C G G G C H H D D D D C C C
```

(e) What is the cache miss rate for each processor?

6808-D:

17/30

6808-T:

16/30

6808-F:

19/30

4 Memory Hierarchy

An enterprising computer architect is building a new machine for high-frequency stock trading and needs to choose a CPU. She will need to optimize her setup for *memory access latency* in order to gain a competitive edge in the market. She is considering two different prototype enthusiast CPUs that advertise high memory performance:

(A) Dragonfire-980 Hyper-Z

(B) Peregrine G-Class XTreme

She needs to characterize these CPUs to select the best one, and she knows from Prof. Mutlu's course that she is capable of reverse-engineering everything she needs to know. Unfortunately, these CPUs are not yet publicly available, and their exact specifications are unavailable. Luckily, important documents were recently leaked, claiming that all three CPUs have:

- Exactly 1 high-performance core
- LRU replacement policies (for any set-associative caches)
- Inclusive caching (i.e., data in a given cache level is present upward throughout the memory hierarchy. For example, if a cache line is present in L1, the cache line is also present in L2 and L3 if available.)
- Constant-latency memory structures (i.e., an access to any part of a given memory structure takes the same amount of time)
- Cache line, size, and associativity are all size aligned to powers of two

Being an ingenious engineer, she devises the following simple application in order to extract all of the information she needs to know. The application uses a high-resolution timer to measure the amount of time it takes to read data from memory with a specific pattern parameterized by ***STRIDE*** and ***MAX_ADDRESS***:

```
start_timer()
repeat N times:
       memory_address <- random_data()
       READ[(memory_address * STRIDE) % MAX_ADDRESS]
end_timer()
```

Assume 1) this code runs for a long time, so all memory structures are fully warmed up, i.e., repeatedly accessed data is already cached, and 2) N is large enough such that the timer captures ***only*** *steady-state information.*

By sweeping ***STRIDE*** and ***MAX_ADDRESS***, the computer architect can glean information about the various memory structures in each CPU.

She produces Figure 1 for CPU A and Figure 2 for CPU B.

Your task: Using the data from the graphs, reverse-engineer the following system parameters. If the parameter *does not make sense* (e.g., L3 cache in a 2-cache system), mark the box with an "X". If the graphs provide *insufficient information* to ascertain a desired parameter, simply mark it as "N/A".

This analysis provides insufficient information to determine the line size of the cache(s). This is because we are always 'striding' in power-of-two values starting at address 0. This means that either our access pattern entirely fits within the cache (in which case we observe constant latency since the cache is already warmed up), or the access pattern is striding using values larger than the line size, so we never see two accesses to the same cache line.

This problem is not actually that hard.
The way to think about these plots is that each point is an access pattern. The easiest points to understand are those that result in an access pattern of {0, 0, 0, 0, ...} and randomly from {0, A}, where A is your stride. Just by looking at those you should be able to determine pretty much everything.
The access latencies and sizes are trivial to read off if you understand what the test code is trying to do. The associativities are nuanced, but you can tell from the aforementioned access patterns by simulating carefully.
If you want to go all-in, you can compute probabilities: if I access {0, A} then 50% of the time I'll hit and 50% miss. It's easy to get the cache latencies, so I can just match points from there on

(a) Fill in the blanks for Dragonfire-980 Hyper-Z.

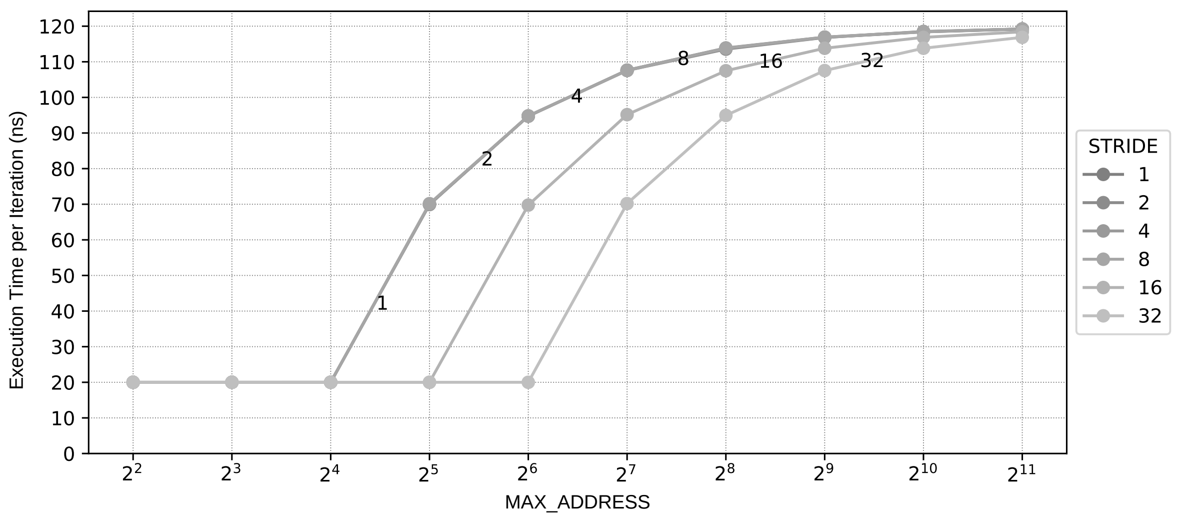


Figure 1: Execution time of the test code on CPU A for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, and 8 overlap in the figure.

Table 1: Fill in the following table for CPU A (Dragonfire-980 Hyper-Z)

System Parameter	**CPU A: Dragonfire-980 Hyper-Z**			
	L1	L2	L3	DRAM
Cache Line Size (B)	N/A	N/A	N/A	N/A OR X
Cache Associativity	2	X	X	X
Total Cache Size (B)	16	X	X	X
Access Latency (ns) [1]	20	X	X	100

[1] e.g., DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.

(b) Fill in the blanks for Peregrine G-Class XTreme.

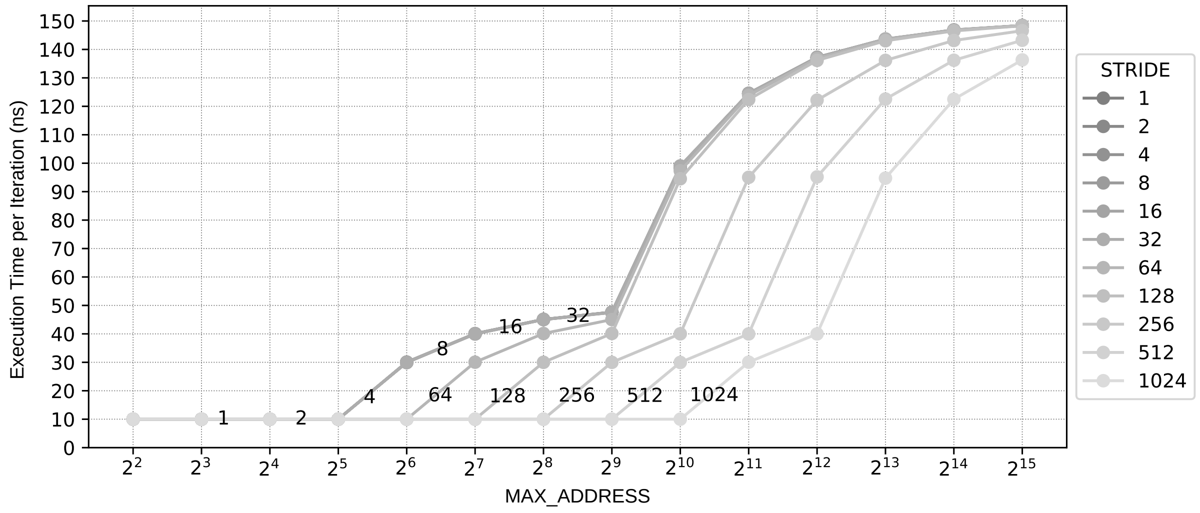


Figure 2: Execution time of the test code on CPU B for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, 8, 16, and 32 overlap in the figure.

Table 2: Fill in the following table for CPU B (Peregrine G-Class XTreme)

System Parameter	**CPU B: Peregrine G-Class XTreme**			
	L1	L2	L3	DRAM
Cache Line Size (B)	N/A	N/A	N/A	N/A OR X
Cache Associativity	1	4	X	X
Total Cache Size (B)	32	512	X	X
Access Latency (ns) [1]	10	40	X	100

[1] e.g., DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.

5 Virtual Memory

An ISA supports an 8-bit, byte-addressable virtual address space. The corresponding physical memory has only 128 bytes. Each page contains 16 bytes. A simple, one-level translation scheme is used and the page table resides in physical memory. The initial contents of the frames of physical memory are shown below.

Frame Number	Frame Contents
0	Empty
1	Page 13
2	Page 5
3	Page 2
4	Empty
5	Page 0
6	Empty
7	Page Table

A three-entry translation lookaside buffer that uses Least Recently-Used (LRU) replacement is added to this system. Initially, this TLB contains the entries for pages 0, 2, and 13. For the following sequence of references, put a circle around those that generate a TLB hit and put a rectangle around those that generate a page fault. What is the hit rate of the TLB for this sequence of references? (Note: LRU policy is used to select pages for replacement in physical memory.)

References (to pages): 0, 13, 5, 2, 14, 14, 13, 6, 6, 13, 15, 14, 15, 13, 4, 3.

References (to pages): (0), (13), 5, 2, [14], (14), 13, [6], (6), (13), [15], 14, (15), (13), [4], [3].
TLB Hit Rate = 7/16

(a) At the end of this sequence, what three entries are contained in the TLB?

4, 13, 3

(b) What are the contents of the 8 physical frames?

Pages 14, 13, 3, 2, 6, 4, 15, Page table

6 Virtual Memory and Caching I

A 2-way set associative write back cache with perfect LRU replacement requires 15×2^9 bits of storage to implement its tag store (including bits for valid, dirty and LRU). The cache is virtually indexed, physically tagged. The virtual address space is 1 MB, page size is 2 KB, cache block size is 8 bytes.

(a) What is the size of the data store of the cache in bytes?

8 KB

The cache is 2-way set associative.
So, each set has 2 tags, each of size t, 2 valid bits, 2 dirty bits and 1 LRU bit (because a single bit is enough to implement perfect LRU for a 2-way set associative cache).
Let i be the number of index bits.

Tag store size $= 2^i \times (2 \times t + 2 + 2 + 1) = 15 \times 2^9$
Therefore,
$2t = 10 \Rightarrow t = 5$
$i = 9$
Data store size $= 2^i \times (2 \times 8)$ bytes $= 2^9 \times (2 \times 8)$ bytes $=$ 8 KB.

(b) How many bits of the virtual index come from the virtual page number?

1 bit

Page size is 2 KB. Hence, the page offset is 11 bits (bits 10:0).
The cache block offset is 3 bits (bits 2:0) and the virtual index is 9 bits (bits 11:3).
Therefore, one bit of the virtual index (bit 11) comes from the virtual page number.

(c) What is the physical address space of this memory system?

64 KB

The page offset is 11 bits.
The physical frame number, which is the same as the physical tag is 5 bits.
Therefore, the physical address space is $2^{(11+5)} = 2^{16}$ bytes $=$ 64 KB.

7 Prefetching I

An architect is designing the prefetch engine for his machine. He first runs two applications A and B on the machine, with a stride prefetcher.
Application A:

```
uint8_t a[1000];
sum = 0;
for (i = 0; i < 1000; i += 4) {
    sum += a[i];
}
```

Application B:

```
uint8_t a[1000];
sum = 0;
for (i = 1; i < 1000; i *= 4) {
    sum += a[i];
}
```

`i` and `sum` are in registers, while the array `a` is in memory. A cache block is 4 bytes in size.

(a) What is the prefetch accuracy and coverage for applications A and B using a stride prefetcher. This stride prefetcher detects the stride between two consecutive memory accesses and prefetches the cache block at this stride distance from the currently accessed block.

Application A's prefetch accuracy is 248/249 and coverage is 248/250.
Application A accesses a[0], a[4], a[8], ... a[996]. It has 1000/4 = 250 accesses to memory.
The first two accesses to a[0] and a[4] are misses. After observing these two accesses, the prefetcher learns that the stride is 4 and starts prefetching a[8], a[12], a[16] and so on until a[1000] (on the access to a[996], a[1000] is prefetched; however, it is not used). In all, 249 cache blocks are prefetched, while only 248 are used.
Hence, the prefetch accuracy is 248/249 and coverage is 248/250.

Application B's prefetch accuracy is 0 and coverage is 0.
Application B accesses a[1], a[4], a[16], a[64] and a[256]. It has five accesses to memory.
However, there isn't a constant stride between these accesses, as the array index is multiplied by 4, rather than being incremented/decremented by a constant stride. Hence, the stride prefetcher cannot prefetch any of the accessed cache blocks and the prefetch accuracy and coverage are both 0.

(b) Suggest a prefetcher that would provide better accuracy and coverage for
i) application A?

A next block prefetcher would always prefetch the next cache block. Hence, the cache block containing a[4] would also be prefetched. Therefore, the prefetch accuracy would be 249/250 and coverage would be 249/250.

ii) application B?

Most common prefetchers such as stride, stream, next block would not improve the prefetch accuracy of application B, as it does not have an access pattern for which prefetchers are commonly designed. Some form of pre-execution, such as runahead execution or dual-core execution could help improve its prefetch accuracy.

(c) Would you suggest using runahead execution for
i) application A. Why or why not?

No. While runahead execution could still provide good prefetch accuracy even for a regular access pattern as in application A, it might not be possible to prefetch all cache blocks before they are accessed, as runahead execution happens only when the application is stalled on a cache miss. A stride prefetcher or next block prefetcher, on the other hand could possibly prefetch all cache blocks before they are accessed, as prefetching happens in parallel with the application's execution.

ii) application B. Why or why not?

Yes. Application B's memory access pattern does not have a regular access stride. Commonly used prefetchers are not designed to prefetch an access pattern like this. However, runahead execution could potentially prefetch some cache blocks, especially as the address of the cache blocks does not depend on the data from a pending cache miss.

8 Prefetching II

Qualtel is designing a next-gen low-power mobile processor codenamed Nemo. You and your colleagues are tasked with designing the prefetcher for Nemo. Nemo has a single core, one level of cache, and a DRAM-based main memory system.

You need to examine different prefetcher designs and analyze the trade-offs involved.

- For all parts of this question, you need to compute the *coverage*, *accuracy* and *bandwidth* over of the prefetcher in its **steady state**.
- If there is a request to a cache block that has gone to main memory, a new request for the same cache block will not go to main memory as the outstanding request has not yet completed. Instead the new request will be coalesced with already outstanding request.

You run an application `libclassical` that has the following memory access pattern (note that these are cache block addresses):

$$A,\ A+1,\ A+2,\ A+7,\ A+8,\ A+9,\ A+14,\ A+15,\ A+16,\ A+21,\ A+22,\ A+23,\ ...$$

Assume this pattern continues for a long time.

(a) You first design a stride prefetcher that observes the last three cache block requests. If there is a constant stride S between the last three requests, the prefetcher issues a prefetch to the next cache block using the stride S. In absence of a constant stride, the prefetcher refrains from prefetching. What is the coverage of your stride prefetcher for `libclassical`? Show your work. Prefetcher coverage is defined as

$$\frac{Total\ number\ of\ correctly\ predicted\ prefetch\ requests}{Total\ number\ of\ unique\ cache\ block\ requests\ without\ the\ prefetcher}$$

0%

Explanation: The prefetcher will learn a constant stride of +1 by seeing, say, cache blocks 14, 15, and 16 and trigger a prefetch to cache block 17. But the prefetched blocks will always be useless; none of the demand requests will be covered by the prefetcher.

(b) What is the the accuracy of your stride prefetcher for `libclassical`? Show your work. Prefetcher accuracy is defined as

$$\frac{Total\ number\ of\ correctly\ predicted\ prefetch\ requests}{Total\ number\ of\ prefetched\ requests}$$

0%

Explanation: None of the prefetched cache blocks will be accurate.

(c) Your colleague designs a new prefetcher that, on a cache block access, prefetches the next N cache blocks. The coverage and accuracy of this prefetcher are 66.67% and 50% respectively for `libclassical`. What is the value of N? Show your work.

$N = 2$

Explanation: For an example, the prefetcher will issue prefetch requests to cache blocks 22 and 23 after seeing the request to 21. Similarly, it will issue prefetches to cache blocks 23 (which will be merged in MSHR) and 24 after seeing request to 22. Hence, every two out of three demand accesses will be covered. And every three out of six prefetch requests will be correct.

(d) The bandwidth overhead of the prefetcher can be defined as

$$\frac{Total\ number\ of\ unique\ cache\ block\ requests\ with\ the\ prefetcher}{Total\ number\ of\ unique\ cache\ block\ requests\ without\ the\ prefetcher}$$

What is the bandwidth overhead of this next-N-block prefetcher for `libclassical`? Show your work.

5/3

Explanation: For every group of three demanded cache blocks, the prefetcher will fetch two additional cache blocks.

(e) What is the minimum value of N required to achieve a 100% prefetch coverage for `libclassical`? Show your work. Remember that you should consider the prefetcher's coverage in its steady state.

$N = 5$

Explanation: In this case, the prefetcher can issue prefetch for, say, cache block 28 by seeing a request to cache block 23

(f) What is the bandwidth overhead at this value of N? Show your work.

7/3

(g) However, you are not happy with the bandwidth overhead required to achieve a prefetch coverage of 100% with a next-N-block prefetcher. You aim to design a prefetcher that achieves a coverage of 100% with a 1× bandwidth overhead. Propose a prefetcher design that accomplishes this goal. Be concrete and clear.

This is an open-ended question.
One solution can be to design a multi-stride prefetcher that can learn a chain of frequently-occurring strides, like $(+1, +1, +5)$. This learning mechanism will cover all accesses without adding any bandwidth overhead.

Extra Exercises for Practicing

The following exercises are old exam questions that are conceptually similar to the ones above, but with slight alterations. We do not expect or recommend you to solve all of them, unless you think you are struggling with a particular concept, or would like to do practice runs on these old exam questions.

9 Cache Structure (Extra)

A byte-addressable system with 16-bit addresses ships with a two-way set associative, writeback cache with perfect LRU replacement. The tag store (including the tag and all other meta-data) requires a total of 4352 bits of storage. What is the block size of the cache? Assume that the LRU information is maintained on a per-set basis as a single bit. (Hint: $4352 = 2^{12} + 2^8$.)

We formulate two basic equations:

$$4352 = 2^{index} * (2 * (tag + dirty + valid) + LRU) \tag{1}$$

$$tag + index + offset = 16 \tag{2}$$

There is one dirty bit and one valid bit per block, and one LRU bit per set. So, now the Equation 1 looks like: $4352 = 2^{index} * (2 * (tag + 2) + 1) = (2^{index} * (2tag + 4)) + 2^{index}$

By using the hint ($4352 = 2^{12} + 2^8$), we get $2^{index} = 2^8$, so $index = 8$, and from $2^{index} * (2tag + 4) = 2^{12}$ we get $(2tag + 4) = 2^4$, so $tag = 6$.

By solving the Equation 2, we get $offset = 2$, so, the block size is $2^2 =$ **4 bytes**

10 Reverse Engineering Caches II (Extra)

Below, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor's data cache:

- Associativity (1, 2 or 4 ways)
- Block size (1, 2, 4, 8, 16, or 32 bytes)
- Total cache size (256 B, or 512 B)
- Replacement policy (LRU or FIFO)

Assumptions: all memory accesses are one byte accesses. All addresses are byte addresses.

Sequence No.	Address Sequence	Hit Ratio
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

Cache block size - 8 bytes
For sequence 1, only 2 out of the 6 accesses (specifically those to addresses 2 and 4) can hit in the cache, as the hit ratio is 0.33. With any other cache block size but 8 bytes, the hit ratio is either smaller or larger than 0.33. Therefore, the cache block size is 8 bytes.

Associativity - 4
For sequence 2, blocks 0, 512, 1024 and 1536 are the only ones that are reused and could potentially result in cache hits when they are accessed the second time. Three of these four blocks should hit in the cache when accessed for the second time to give a hit rate of 0.33 (3/9).
Given that the block size is 8 and for either cache size (256B or 512B), all of these blocks map to set 0. Hence, an associativity of 1 or 2 would cause at most one or two of these four blocks to be present in the cache when they are accessed for the second time, resulting in a maximum possible hit rate of less than 3/9. However, the hit rate for this sequence is 3/9. Therefore, an associativity of 4 is the only one that could potentially give a hit rate of 0.33 (3/9).

Total cache size - 256 B
For sequence 3, a total cache size of 512 B will give a hit rate of 4/9 with a 4-way associative cache and 8 byte blocks regardless of the replacement policy, which is higher than 0.33. Therefore, the total cache size is 256 bytes.

Replacement policy - LRU
For the aforementioned cache parameters, all cache lines in sequence 4 map to set 0. If a FIFO replacement policy were used, the hit ratio would be 3/8, whereas if an LRU replacement policy were used, the hit ratio would be 1/4. Therefore, the replacement policy is LRU.

11 Cache Performance Analysis (Extra)

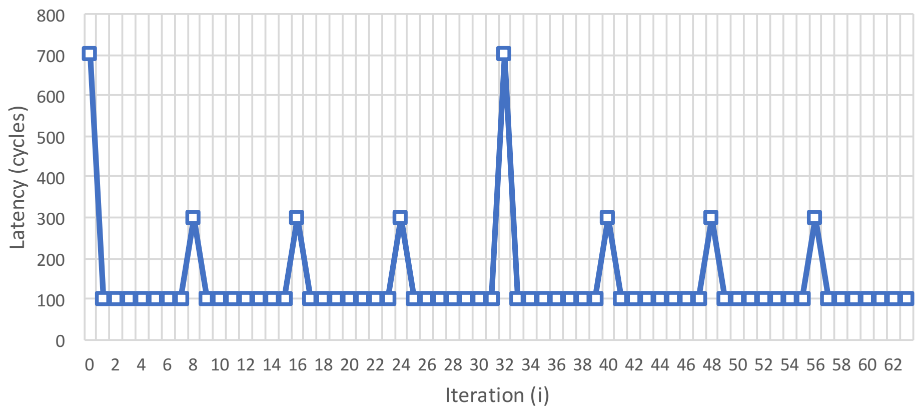
We are going to microbenchmark the cache hierarchy of a computer with the following two codes. The array `data` contains 32-bit unsigned integer values. For simplicity, we consider that accesses to the array `latency` bypass all caches (i.e., `latency` is *not* cached). `timer()` returns a timestamp in cycles.

```
(1) j = 0;
     for (i=0; i<size; i+=stride){
        start = timer();
        d = data[i];
        stop = timer();
        latency[j++] = stop - start;
     }

(2) for (i=0; i<size1; i+=stride1){
        d = data[i];
     }
     j = 0;
     for (i=0; i<size2; i+=stride2){
        start = timer();
        d = data[i];
        stop = timer();
        latency[j++] = stop - start;
     }
```

The cache hierarchy has two levels. L1 is a 4kB set associative cache.

(a) When we run code (1), we obtain the latency values in the following chart for the first 64 reads to the array `data` (in the first 64 iterations of the loop) with `stride` equal to 1. What are the cache block sizes in L1 and L2?



L1 cache block size is 32 bytes, and L2 cache block size is 128 bytes.

Explanation:
The highest latency values (700 cycles) correspond to L2 cache misses. The latency of 300 cycles correspond to L1 cache misses that hit the L2 cache. The lowest latency (100 cycles) corresponds to L1 cache hits. We find L1 misses every 8 32-bit elements, and L2 misses every 32 32-bit elements. Thus, L1 cache blocks are of size 32 bytes, while L2 cache blocks are 128 bytes.

(b) Using code (2) with `stride1` = `stride2` = 32, `size1` = 1056, and `size2` = 1024, we observe `latency[0]` = 300 cycles. However, if `size1` = 1024, `latency[0]` = 100 cycles. What is the maximum number of ways in L1? (Note: The replacement policy can be either FIFO or LRU).

The maximum number of ways is 32.

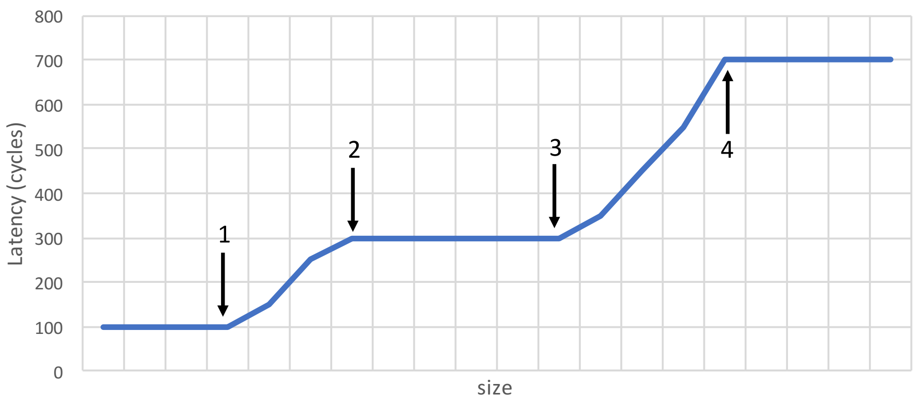
Explanation:
In L1 there are a total of 128 cache blocks. If the accessed space is 1056 elements, 33 cache blocks are read. In case of having more than 32 ways, there would be a hit in the second loop. However, with 32 or less ways, the cache block that contains `data[0]` is replaced in the last iteration of the first loop.

(c) We want to find out the exact replacement policy, assuming that the associativity is the maximum obtained in part (b). We first run code (2) with `stride1` = 32, `size1` = 1024, `stride2` = 64, and `size2` = 1056. Then (after resetting `j`), we run code (1) with `stride` = 32 and `size` = 1024. We observe `latency[1]` = 100 cycles. What is the replacement policy? Explain. (Hint: The replacement policy can be either FIFO or LRU. You need to find the correct one and explain).

It is FIFO.

Explanation:
In the second loop of code (2), the last cache block that is accessed (`data[1024]`) replaces the cache block that contains `data[0]`, if the replacement policy is FIFO. If the replacement policy is LRU, the LRU cache block (the one that contains `data[32]`). Because we observe that `latency[1]` is 100 cycles, and it corresponds to the access to `data[32]`, we conclude the replacement policy is FIFO.

(d) Now we carry out two consecutive runs of code (1) for different values of `size`. In the first run, `stride` is equal to 1. In the second run, `stride` is equal to 16. We ignore the latency results of the first run, and average the latency results of the second run. We obtain the following graph. What do the four parts shown with the arrows represent?



Before arrow 1:

The entire array fits in L1. In arrow 1 the size of the array is the same as the size of L1 (4kB).

Between arrow 1 and arrow 2:

Some accesses in the second run hit in L1 and other accesses hit in L2.

Between arrow 2 and arrow 3:

All accesses in the second run hit in L2.

Between arrow 3 and arrow 4:

Still some accesses in the second run hit in L2. Arrow 3 marks the size of the L2 cache.

After arrow 4:

The accesses of the second run always miss in L2, and it is necessary to access main memory.

Explain as needed (if you need more):

12 Reverse Engineering Caches III (Extra)

You are trying to reverse-engineer the characteristics of a cache in a system, so that you can design a more efficient, machine-specific implementation of an algorithm you are working on. To do so, you have come up with three sequences of memory accesses to various *bytes* in the system in an attempt to determine the following four cache characteristics:

- Cache block size (8, 16, 32, 64, or 128 B).
- Cache associativity (2-, 4-, or 8-way).
- Cache replacement policy (LRU or FIFO).
- Cache size (4 or 8 KiB).

The only statistic that you can collect on this system is *cache hit rate* after performing each sequence of memory accesses. Here is what you observe:

Sequence	Addresses Accessed (Oldest → Youngest)								Hit Rate
1.	0	16	24	25	1024	255	1100	305	2/8
2.	31	65536	65537	131072	262144	8	305	1060	3/8
3.	262145	65536	4						2/3

Assume that the cache is initially empty at the beginning of the first sequence, but *not* at the beginning of the second and third sequence. The sequences are executed back-to-back, i.e., no other accesses take place in between sequences. Thus, **at the beginning of the second sequence, the contents are the same as at the end of the first sequence. At the beginning of the third sequence, the contents are the same as at the end of the second sequence**.

Based on what you observe, what are the following characteristics of the cache? Explain to get points.

(a) [20 points] Cache block size (8, 16, 32, 64, or 128 B)?

16 B.

Explanation:
Cache hit rate is 2/8 in sequence 1. This means that there are 2 hits. Depending on the cache block size, we can group addresses that belong to the same cache block as follows:

- **8B:** {0}, {16}, {24,25}, {255}, {305}, {1024}, {1100}. ∴ Number of possible hits = 1.
- **16B:** {0}, {16,24,25}, {255}, {305}, {1024}, {1100}. ∴ Number of possible hits = 2.
- **32B:** {0,16,24,25}, {255}, {305}, {1024}, {1100}. ∴ Number of possible hits = 3.
- **64B:** {0,16,24,25}, {255,305}, {1024}, {1100}. ∴ Number of possible hits = 4.
- **128B:** {0,16,24,25}, {255,305}, {1024,1100}. ∴ Number of possible hits = 5.

Therefore, we can know that the cache block size is 16B.

(b) [20 points] Cache associativity (2-, 4-, or 8-way)?

2-way.

Explanation:

Cache hit rate is 3/8 in sequence 2, which means that there are 3 hits.
We already know that the cache block size is 16B. Thus, there are 4 offset bits.

The access to address 31 in sequence 2 would hit because the cache block would not be replaced.
The access to address 305 in sequence 2 would hit because the cache block would not be replaced.
The access to address 65537 in sequence 2 would hit because the cache block would not be replaced.

Therefore, all the other accesses should miss.
The access to address 65536, 131072 and 262144 in sequence 2 would miss because addresses 65536, 131072 and 262144 do not belong to any cache block previously accessed.
Addresses 65536, 131072 and 262144 would be placed in set 0 if the cache associativity is 2-way, 4-way, or 8-way, independently of the cache size.
Address 8 must be a miss, so its cache block must be replaced by cache blocks that map to set 0 (addresses 65536, 131072 and 262144). For this to happen, the associativity must be 2-way.
Therefore, the cache is 2-way associative.

(c) [20 points] Cache replacement policy (LRU or FIFO)?

FIFO.

Explanation:
From questions (a) and (b), we already know the following facts:

- The cache block size is 16 B.
- The cache is 2-way.

Cache hit rate is 2/3 in sequence 3, which means that there are 2 hits.
With the LRU policy only the access to address 262145 in sequence 3 would hit. With the FIFO policy, accesses to addresses 262145 and 4 in sequence 3 would hit.
Therefore, the cache adopts the FIFO policy.

(d) [10 points] To identify the cache size (4 or 8KiB), you can access two addresses right after sequence 3 (i.e., the contents are the same as at the end of the third sequence) and measure the cache hit rate. Which two addresses would you choose? Explain your answer (there may be several correct answers).

Address 2048 and address 0 (there are other correct answers as well)

Explanation:
From questions (a), (b) and (c), we already know the following facts:

- The cache block size is 16 B.
- The cache is 2-way.
- FIFO replacement policy

We know that there are 4 bits for indexing the byte in a block, and there are 7 bits (if the cache size is 4KiB) or 8 bits (if the cache size is 8 KiB). Therefore, address 2048 would be in set 0 only if the cache size is 4KiB: we can access address 2048, and then check if a block in set 0 was replaced by address 2048 by accessing address 0. If it is a miss, the cache size is 4KiB, and if it is a hit, the cache size is 8KiB.

13 Reverse Engineering Caches IV (Extra)

You are trying to reverse-engineer the characteristics of a cache in a system, so that you can design a more efficient, machine-specific implementation of an algorithm you are working on. To do so, you have come up with three patterns that access various *bytes* in the system in an attempt to determine the following four cache characteristics:

- Cache block size (8, 16, 32, 64, or 128 B)
- Cache associativity (1-, 2-, 4-, or 8-way)
- Cache size (4 or 8 KB)
- Cache replacement policy (LRU or FIFO)

However, the only statistic that you can collect on this system is *cache hit rate* after performing the access pattern. Here is what you observe:

Sequence	Addresses Accessed (Oldest → Youngest)								Hit Rate
1.	0	4	8	16	64	128			1/2
2.	31	8192	63	16384	4096	8192	64	16384	5/8
3.	32768	0	129	1024	3072	8192			1/3

Assume that the cache is initially empty at the beginning of the first sequence, but not at the beginning of the second and third sequences. The sequences are executed back-to-back, i.e., no other accesses take place between the three sequences. Thus, **at the beginning of the second (third) sequence, the contents are the same as at the end of the first (second) sequence**.

Based on what you observe, what are the following characteristics of the cache?

(a) Cache block size (8, 16, 32, 64, or 128 B)?

64 B.

Explanation:
Cache hit rate is 1/2 in sequence 1. This means that there are 3 hits, which are necessarily in addresses 4, 8, and 16. Thus, cache block size might be 32 or 64 B.

In sequence 2, there are only three misses, which are in addresses 8192, 16384, and 4096. The remaining 5 accesses are hits. For 63 to be a hit, the cache block size should be 64 B.

(b) Cache associativity (1-, 2-, 4-, or 8-way)?

4-way.

Explanation:
We already know that the cache block size is 64 B. Thus, there are 6 offset bits.

If 1-way, 63 would miss, since 8192 would map to the same set regardless of cache size (i.e., bits 6 to 12 are equal).

Addresses 0, 4096, 8192, 16384, and 32768 map to the same set. If 2-way, the second access to 8192 and 16384 (in sequence 2) would not hit.

If 8-way, 1024 and 3072 would map to the same set as 0, 4096, 8192, 16384, and 32768, since they all share bits 6 to 9. In that case, 0 and 8192 would be both hits in sequence 3. This is not possible because there are only two hits in sequence 3. 32768, 1024, and 3072 are compulsory misses, while 129 is a hit (address 128 was accessed by sequence 1). Thus, either 0 or 8192 should miss in sequence 3.

(c) Cache size (4 or 8 KB)?

8 KB.

Explanation:
We know that the cache is 4-way associative. The access to address 0 in sequence 3 is a miss, because the cache block was replaced by address 32768. Thus, access to 8192 should be a hit.

If 4-way and 4 KB, 1024 and 3072 would map to the same set as 8192. In that case, 8192 would miss. So the size of the cache should be 8 KB.

(d) Cache replacement policy (LRU or FIFO)?

LRU.

Explanation:
For 8192 to hit in sequence 3, 4096 should have been replaced by 0. So the replacement policy is LRU, because FIFO would have replaced 8192.

14 Virtual Memory and Caching II (Extra)

A four-way set-associative writeback cache has a 2^{11} * 89-bit tag store. The cache uses a custom replacement policy that requires 9 bits per set. The cache block size is 64 bytes. The cache is virtually-indexed and physically-tagged. Data from a given physical address can be present in up to eight different sets in the cache. The system uses hierarchical page tables with two levels. Each level of the page table contains 1024 entries. A page table may be larger or smaller than one page. The TLB contains 64 entries.

(a) How many bits of the virtual address are used to choose a set in the cache?

tag store: 89 bits = (9 replacement + 4 dirty + 4 valid) + 4 * (18 physical tag bits)
$2^{11} \rightarrow$ 2K sets $\rightarrow$ 8K blocks $\rightarrow$ 512KB cache
11

(b) What is the size of the cache data store?

512KB

(c) How many bits in the Physical Frame Number must overlap with the set index bits in the virtual address?

3

(d) On the following blank figure representing a virtual address, draw in bitfields and label bit positions for *cache block offset* and *set number*. Be complete, showing the beginning and ending bits of each field.

Virtual Address:

16	6	5	0
Index		Block offset	

(e) On the following blank figure representing a physical address, draw in bitfields and label bit positions for *physical frame number* and *page offset*. Be complete, showing the beginning and ending bits of each field.

Physical Address:

31	14	13	0
PFN		Page offset	

(f) What is the page size?

16KB

(g) What is the size of the virtual address space?

16GB

(h) What is the size of the physical address space?

4GB

15 Prefetching III (Extra)

A runahead execution processor is designed with an unintended hardware bug: every other instruction in runahead mode is dropped by the processor after the fetch stage. Recall that the runahead mode is the speculative processing mode where the processor executes instructions solely to generate prefetch requests. All other behavior of the runahead mode is exactly as we described in lectures. When a program is executed, which of the following scenarios could happen compared to a runahead processor without the hardware bug and why? Circle YES if there is a possibility to observe the described behavior and explain in the box (either if you answer YES or NO). Assume that the program has no bug in it and executes correctly on the processor without the hardware bug.

(a) [8 points] The buggy runahead processor finishes the program *correctly* and *faster* than the non-buggy runahead processor. Why?

[✓ *YES*] *NO*

Dropping instructions enables the discovery of more cache misses than not dropping the instructions.

(b) [8 points] The buggy runahead processor finishes the program *correctly* and *slower* than the non-buggy runahead processor. Why?

[✓ *YES*] *NO*

The buggy runahead processor is not able to generate cache misses that are dependent on dropped instructions.

(c) [9 points] The buggy runahead processor executes the program *incorrectly.* Why?

YES [✓ *NO*]

Not possible as all executions in runahead mode is purely speculative and do not commit. Hence it cannot affect the correctness of the program.