Design of Digital Circuits (252-0028-00L), Spring 2019
Optional HW 7: Caches and Virtual Memory
SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Mohammed Alser, Can Firtina, Hasan Hassan, Juan Gomez Luna, Lois Orosa, Giray Yaglikci

Released: Friday, May 31, 2019

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 is 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.

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).

(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:
CPI = 2 + 0.33×10 + 0.33×0.25×10 = 6.1250 cycles, if `X` and `Y` are different arrays.
CPI = 2 + 0.33×10 + 0.33×0.125×10 = 5.7125 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 $= 2 + 0.25\times10 + 0.50\times0.25\times10 = 5.75$ cycles, if `X` and `Y` are different arrays.
CPI $= 2 + 0.25\times10 + 0.50\times0.125\times10 = 5.125$ 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 Reverse Engineering Caches II

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? Explain to get points.

(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.

4 Analyzing Cache Structure

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.

5 Caches

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**

6 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".

NOTE 1 TO SOLUTION READER:
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.

NOTE 2 TO SOLUTION READER:
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.

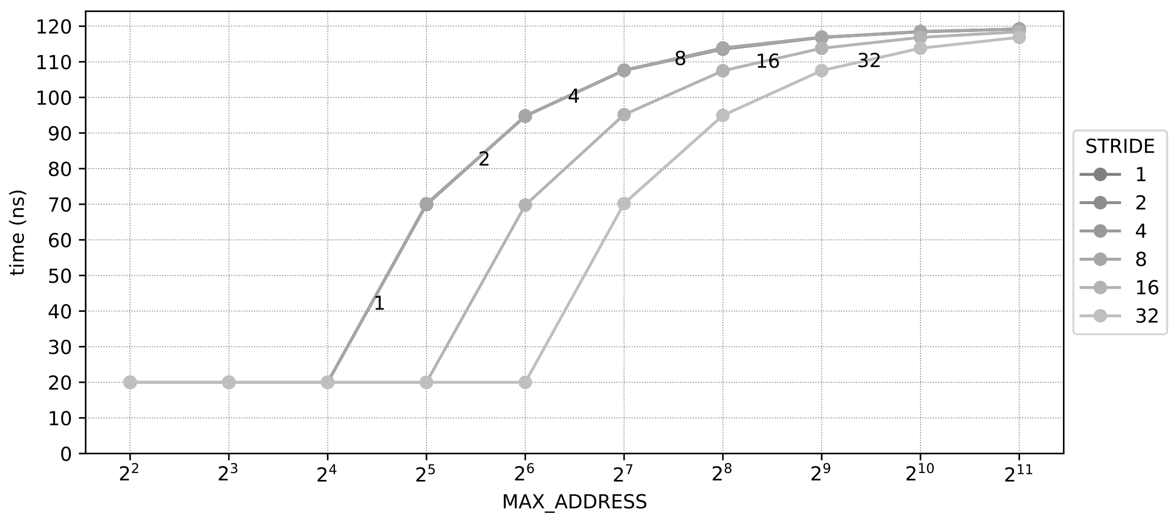

Figure 1: Execution time of the test code on CPU A for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, and 8 overlap in the figure.

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.

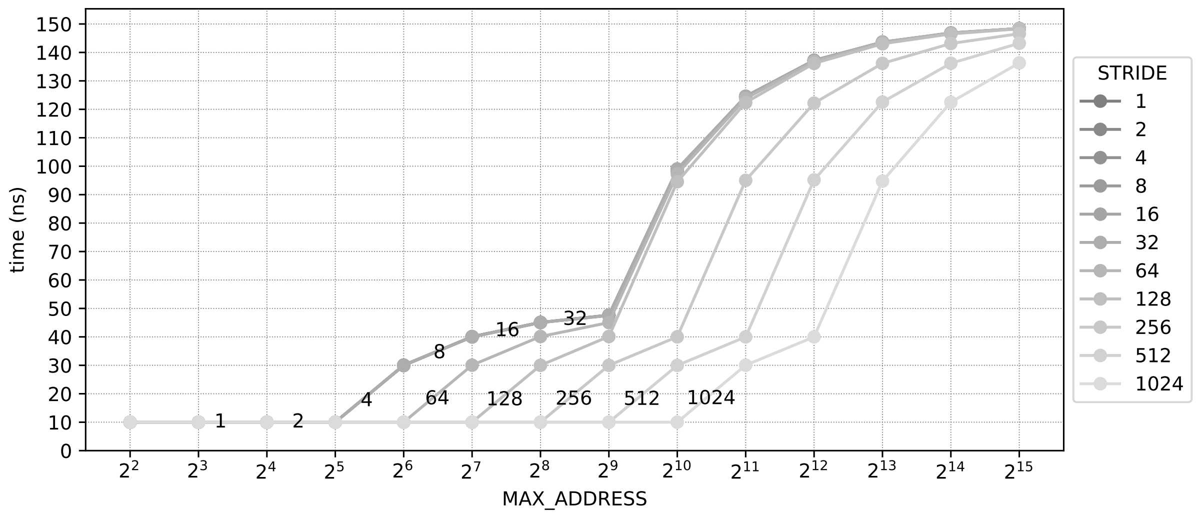

Figure 2: Execution time of the test code on CPU B for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, 8, 16, and 32 overlap in the figure.

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.

7 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