

Name:

Student ID:

Final Exam
Computer Architecture (263-2210-00L)
ETH Zürich, Fall 2017

Prof. Onur Mutlu

Problem 1 (80 Points):	
Problem 2 (80 Points):	
Problem 3 (60 Points):	
Problem 4 (65 Points):	
Problem 5 (90 Points):	
Problem 6 (60 Points):	
Problem 7 (65 Points):	
Problem 8 (BONUS: 45 Points):	
Problem 9 (BONUS: 75 Points):	
<hr/>	
Total (620 (500 + 120 bonus) Points):	

Examination Rules:

1. Written exam, 180 minutes in total.
2. No books, no calculators, no computers or communication devices. 6 pages of handwritten notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam.
4. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
5. Put your Student ID card visible on the desk during the exam.
6. If you feel disturbed, immediately call an assistant.
7. Write with a black or blue pen (no pencil, no green or red color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake.
9. Please write your initials at the top of every page.

Tips:

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You may be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.
- **Write legibly.** Show your final answer.

Initials: _____

Computer Architecture

January 29th, 2018

This page intentionally left blank

1 DRAM Refresh [80 points]

A memory system is composed of eight banks, and each bank contains 32K rows. The row size is 8 KB.

Every DRAM row refresh is initiated by a command from the memory controller, and it refreshes a single row. Each refresh command keeps the command bus busy for 5 ns.

We define *command bus utilization* as a fraction of the total time during which the command bus is busy due to refresh.

The retention time of each row depends on the temperature (T). The rows have different retention times, as shown in the following Table 1:

Retention Time	Number of rows
$(128 - T)$ ms, $0^\circ C \leq T \leq 128^\circ C$	2^8 rows
$2 * (128 - T)$ ms, $0^\circ C \leq T \leq 128^\circ C$	2^{16} rows
$4 * (128 - T)$ ms, $0^\circ C \leq T \leq 128^\circ C$	all other rows
$8 * (128 - T)$ ms, $0^\circ C \leq T \leq 128^\circ C$	2^8 rows

Table 1: Retention time

1.1 Refresh Policy A [20 points]

Assume that the memory controller implements a refresh policy where all rows are refreshed with a fixed refresh interval, which covers the worst-case retention time (Table 1).

- (a) [5 points] What is the maximum temperature at which the DRAM can operate reliably with a refresh interval of 32 ms?

- (b) [15 points] What command bus utilization is directly caused by DRAM refreshes (with refresh interval of 32 ms)?

1.2 Refresh Policy B [15 points]

Now assume that the memory controller implements a refresh policy where all rows are refreshed only *as frequently as required* to correctly maintain their data (Table 1).

- (a) [15 points] How many refreshes are performed by the memory controller during a 1.024 second period? (with $T=64^{\circ}C$)

1.3 Refresh Policy C [25 points]

Assume that the memory controller implements an even smarter policy to refresh the memory. In this policy, the refresh interval is fixed, and it covers the worst-case retention time (64ms), as the refresh policy in part 1.1. However, as an optimization, a row is refreshed only if it has *not* been **accessed** during the past refresh interval. For maintaining correctness, if a cell reaches its maximum retention time without being refreshed, the memory controller issues a refresh command.

- (a) [5 points] Why does a row *not* need to be refreshed if it was accessed in the past refresh interval?

- (b) [20 points] A program accesses all the rows repeatedly in the DRAM. The following table shows the access interval of the rows, the number of rows accessed with the corresponding access interval, and the retention times of the rows that are accessed with the corresponding interval.

Access interval	Number of rows	Retention times
1ms	2^{16} rows	64ms, 128ms or 256ms
60ms	2^{16} rows	64ms, 128ms or 256ms
128ms	all other rows	128ms or 256ms

What command bus utilization is directly caused by DRAM refreshes?

1.4 Refresh Policy D [20 points]

Assume that the memory controller implements an approximate mechanism to reduce refresh rate using Bloom filters, as we discussed in class. For this question we assume the retention times in Table 1 with a constant temperature of $64^{\circ}C$.

One Bloom filter is used to represent the set of all rows that require a 64 ms refresh rate.

The memory controller's refresh logic is modified so that on every potential refresh of a row (every 64 ms), the refresh logic probes the Bloom filter. If the Bloom filter probe results in a "hit" for the row address, then the row is refreshed. Any row that does *not* hit in the Bloom filter is refreshed at the default rate of once per 128 ms.

(a) [20 points] The memory controller performs 2107384 refreshes in total across the channel over a time interval of 1.024 seconds. What is the false positive rate of the Bloom filter? (NOTE: False positive rate = Total number of false positives / Total number of accesses).

- Hint: $2107384 = 2^3 * (2^{18} + 2^{11} - 2^{10} + 2^9 - 2^8 - 1)$

2 DRAM Scheduling and Latency [80 points]

You would like to understand the configuration of the DRAM subsystem of a computer using reverse engineering techniques. Your current knowledge of the particular DRAM subsystem is limited to the following information:

- The physical memory address is 16 bits.
- The DRAM subsystem consists of a single channel and 4 banks.
- The DRAM is byte-addressable.
- The most-significant 2 bits of the physical memory address determine the bank.
- The DRAM command bus operates at 500 MHz frequency.
- The memory controller issues commands to the DRAM in such a way that *no command* for servicing a *later* request is issued before issuing a READ command for the current request, which is the oldest request in the request buffer. For example, if there are requests A and B in the request buffer, where A is the older request and the two requests are to different banks, the memory controller does *not* issue an ACTIVATE command to the bank that B is going to access *before* issuing a READ command to the bank that A is accessing.

You realize that you can observe the memory requests that are waiting to be serviced in the request buffer. At a particular point of time, you take the snapshot of the request buffer and you observe the following requests in the request buffer.

Requests in the request buffer (in descending order of request age, where the oldest request is on the top):

time	↓	Read 0x4C80
		Read 0x0140
		Read 0x4EC0
		Read 0x8000
		Read 0xF000
		Read 0x803F
		Read 0x4E80
		↓

At the same time you take the snapshot of the request buffer, you start probing the DRAM command bus. You observe the DRAM command type and the cycle (relative to the first command) at which the command is seen on the DRAM command bus. The following are the DRAM commands you observe on the DRAM bus while the requests above are serviced.

```
Cycle 0 --- PRECHARGE
Cycle 6 --- ACTIVATE
Cycle 10 --- READ
Cycle 11 --- READ
Cycle 21 --- PRECHARGE
Cycle 27 --- ACTIVATE
Cycle 31 --- READ
Cycle 32 --- ACTIVATE
Cycle 36 --- READ
Cycle 37 --- READ
Cycle 38 --- READ
Cycle 42 --- PRECHARGE
Cycle 48 --- ACTIVATE
Cycle 52 --- READ
```

Answer the following questions using the information provided above.

- (a) [15 points] What are the following DRAM timing parameters used by the memory controller, in terms of nanoseconds?

i) ACTIVATE-to-READ latency

ii) ACTIVATE-to-PRECHARGE latency

iii) PRECHARGE-to-ACTIVATE latency

- (b) [20 points] What is the row size in bytes? Explain your answer.

- (c) [20 points] What is the status of the banks *prior* to the execution of any of the the above requests? In other words, which rows from which banks were open immediately prior to issuing the DRAM commands listed above? Fill in the table below indicating whether a bank has an open row, and if there is an open row, specify its address. If there is not enough information to infer the open row address, write *unknown*.

	Open or Closed?	Open Row Address
Bank 0		
Bank 1		
Bank 2		
Bank 3		

- (d) [25 points] To improve performance, you decide to implement the idea of Tiered-Latency DRAM (TL-DRAM) in the DRAM chip. Assume that a bank consists of a single subarray. With TL-DRAM, an entire bank is divided into a near-segment and far-segment. When accessing a row in the near-segment, the ACTIVATE-to-READ latency *reduces* by 2 cycles and the ACTIVATE-to-PRECHARGE latency reduces by 5 cycles. When accessing a row in the far-segment, the ACTIVATE-to-READ latency *increases* by 1 cycle and the ACTIVATE-to-PRECHARGE latency increases by 2 cycles.

Assume that the rows in the near-segment have smaller row ids compared to the rows in the far-segment. In other words, physical memory row addresses 0 through $N - 1$ are the near-segment rows, and physical memory row addresses N through $M - 1$ are the far-segment rows.

If the above DRAM commands are issued 5 cycles faster with TL-DRAM compared to the baseline (the last command is issued in cycle 47), how many rows are in the near-segment? Show your work.

3 Cache Reverse Engineering [60 points]

Consider a processor using a 4-block LRU-based L1 data cache. Starting with an empty cache, an application accesses three L1 cache blocks in the following order, where consecutive numbers (e.g., n , $n + 1$, $n + 2$, ...) represent the starting addresses of consecutive cache blocks in memory:

$$n \rightarrow n + 2 \rightarrow n + 4$$

3.1 Part I: Vulnerability [35 points]

A malicious programmer realizes she can reverse engineer the number of sets and ways in the L1 data cache by issuing *just* two more accesses and observing *only* the cache hit rate across these two accesses. Assume that she can insert the malicious accesses only after the above three accesses of the program.

- [15 points] What are the next two cache blocks she should access? (e.g., $[n+?, n+?]$)

- [10 points] How many L1 sets/ways would there be if the cache hit rate over the 2 extra instructions was:

L1 hit rate	# sets	# ways
100%		
50%		
0%		

- [10 points] What should the next two accesses be if the replacement policy had been Most Recently Used (MRU)? (e.g., $[n+?, n+?]$)

3.2 Part II: Exploitation [25 points]

Assuming the original cache design (i.e., with an LRU replacement policy) is using a 1-set (4-way) configuration, the malicious programmer decides to disrupt a high-security banking application, allowing her to transfer large amounts of foreign assets into her own secure Swiss account. By using a carefully designed concurrently running process to interfere with the banking application, she would like to induce slowdown, thereby exposing opportunity for her mischief.

Assume that the unmodified banking application issues the following access pattern, where each number represents X in $n + X$:

0 → 6 → 1 → 7 → 6 → 1 → 4 → 0 → 5 → 0 → 7 → 4 → 2 → 7 → 2 → 4

1. [10 points] What is the unmodified banking application's cache hit rate?

2. [15 points] Now, assume that the malicious programmer knows the access pattern of the banking application. Using this information, she is able to inject a **single** extra cache access in between each of the banking application's accesses (i.e., she can interleave a malicious access pattern with the normal application's execution).

What is the minimum cache hit rate (not counting extra malicious accesses) that the malicious programmer can induce for the banking application?

4 Runahead Execution [65 points]

Assume an in-order processor that employs Runahead execution, with the following specifications:

- The processor enters Runahead mode when there is a cache miss.
- There is no penalty for entering and leaving the Runahead mode.
- There is a 64KB data cache. The cache block size is 64 bytes.
- Assume that the instructions are fetched from a separate dedicated memory that has zero access latency, so an instruction fetch never stalls the pipeline.
- The cache is 4-way set associative and uses the LRU replacement policy.
- A memory request that hits in the cache is serviced instantaneously.
- A cache miss is serviced from the main memory after X cycles.
- A cache block for the corresponding fetch is allocated *immediately* when a cache miss happens.
- The cache replacement policy does *not* evict the cache block that triggered entry into Runahead mode until after the Runahead mode is exited.
- The victim for cache eviction is picked at the same time a cache miss occurs, i.e., during cache block allocation.
- ALU instructions and Branch instructions take one cycle.
- Assume that the pipeline *never stalls* for reasons *other than data cache misses*. Assume that the conditional branches are always correctly predicted and the data dependencies do not cause stalls (except for data cache misses).

Consider the following program. Each element of Array A is one byte.

```
for(int i=0;i<100;i++){ \\ 2 ALU instructions and 1 branch instruction
    int m = A[i*16*1024]+1; \\ 1 memory instruction followed by 1 ALU instruction
    ... \\ 26 ALU instructions
}
```

- (a) [20 points] After running this program using the processor specified above, you find that there are 66 data cache hits. What are **all** the possible values of the cache miss latency X ? You can specify all possible values of X as an inequality. Show your work.

- (b) [20 points] Is it possible that *every* memory access in the program misses in the cache? If so, what are **all** possible values of X that will make all memory accesses in the program miss in the cache? If not, why? Show your work.

- (c) [25 points] What is the *minimum* number of cache misses that the processor can achieve by executing the above program? Show your work.

5 Interconnection Networks [90 points]

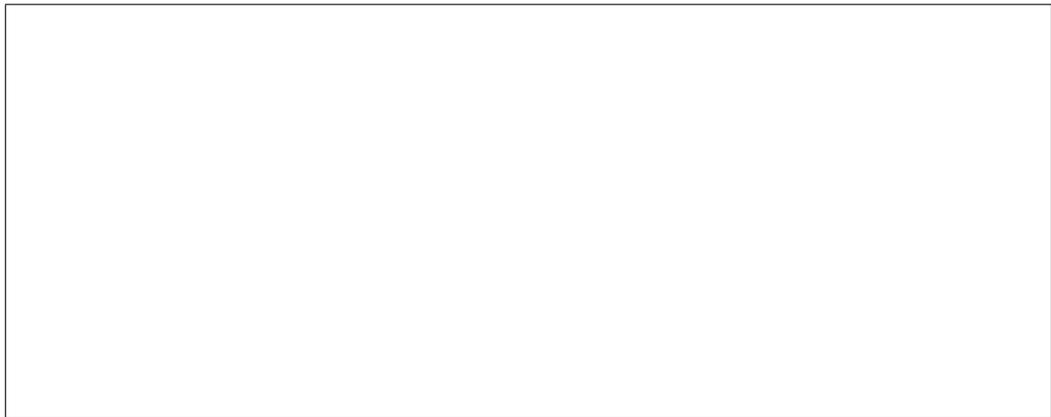
Suppose you would like to connect 2^N processors, and you are considering four different topologies:

- $\sqrt{2^N} \times \sqrt{2^N}$ 2D mesh
- $\sqrt{2^{N-2}} \times \sqrt{2^{N-2}}$ 2D concentrated mesh (Cmesh), where each router serves four processors
- $\sqrt{2^N} \times \sqrt{2^N}$ 2D torus
- Hypercube

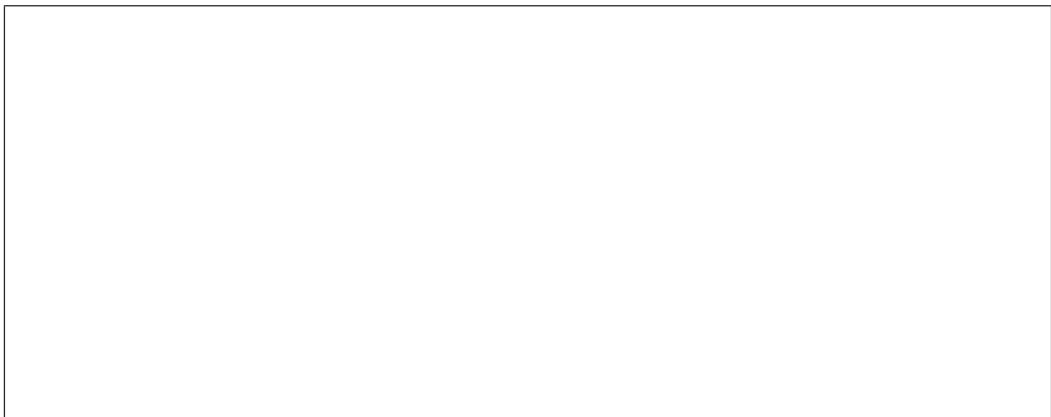
Please answer the following questions. Show your work.

- (a) [20 points] For $N = 4$, please draw how each network looks like. You can use ... (three dots) to avoid repeated patterns.

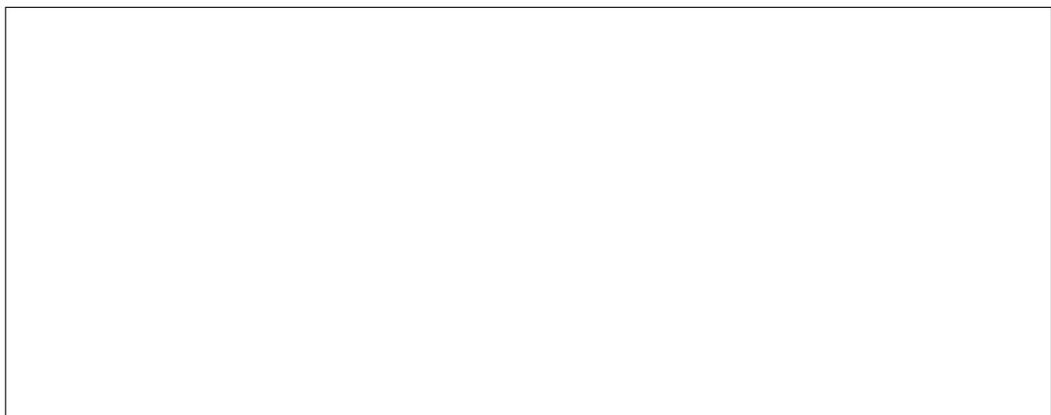
2D mesh



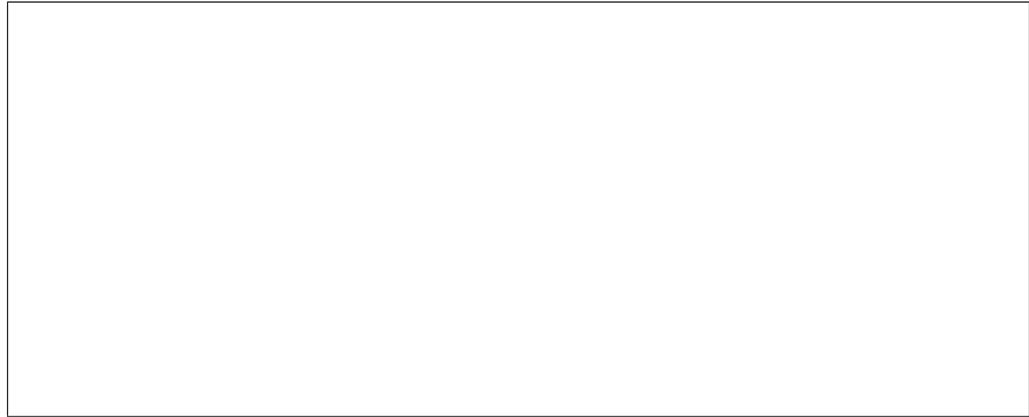
Cmesh



2D torus



Hypercube



For the remaining questions, *assume* $N = 8$.

- (b) [20 points] For $N = 8$, calculate the number of network links for each network. (Hint: a single network link is bi-directional)



- (c) [25 points] For $N = 8$, calculate the number of input/output ports including the injection/ejection ports for *each router* in these topologies (Hint: give answer to all types of routers that exist in an irregular network).



- (d) [25 points] Assume a network link can be faulty. For each topology, what is the minimum possible number of faulty links that are needed to make at least one processor unreachable from any other processor?



6 Cache Coherence [60 points]

We have a system with 4 byte-addressable processors. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. Coherence is maintained using the Illinois Protocol (MESI), which sends an invalidation to other processors on writes, and the other processors invalidate the block in their caches if *the block is present* (NOTE: On a write hit in one cache, a cache block in Shared state becomes Modified in that cache).

Accessible memory addresses range from 0x50000000 – 0x5FFFFFFF. Assume that the offset within a cache block is 0 for all memory requests. We use a snoopy protocol with a shared bus.

Cosmic rays strike the MESI state storage in your coherence modules, causing the state of a *single* cache line to instantaneously change to another state. This change causes an inconsistent state in the system. We show below the initial tag store state of the four caches, *after* the inconsistent state is induced.

Initial State

Cache 0		
	Tag	MESI state
Set 0	0x5FFFFFFF	M
Set 1	0x5FFFFFFF	E
Set 2	0x5FFFFFFF	S
Set 3	0x5FFFFFFF	I

Cache 2		
	Tag	MESI state
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFFFF	S
Set 3	0x533333	S

Cache 1		
	Tag	MESI state
Set 0	0x522222	I
Set 1	0x510000	S
Set 2	0x5FFFFFFF	S
Set 3	0x533333	S

Cache 3		
	Tag	MESI state
Set 0	0x5FF000	E
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

(a) [15 points] What is the inconsistency in the above initial state? Explain with reasoning.

(b) [20 points] Consider that, after the initial state, there are several paths that the program can follow that access different memory instructions. In b.1-b.4, we will examine whether the followed path can potentially lead to incorrect execution, i.e., an incorrect result.

b.1) [10 points] Could the following path potentially lead to incorrect execution? Explain.

order	Processor 0	Processor 1	Processor 2	Processor 3
1 st			ld 0x51110040	
2 nd	st 0x5FFFFFF40			
3 rd				st 0x51110040
4 th		ld 0x5FFFFFF80		
5 th		ld 0x51110040		
6 th		ld 0x5FFFFFF40		

b.2) [10 points] Could the following path potentially lead to incorrect execution? Explain.

order	Processor 0	Processor 1	Processor 2	Processor 3
1 st				ld 0x51110040
2 nd	ld 0x5FFFFFF00			
3 rd			ld 0x51234540	
4 th	st 0x5FFFFFF40			
5 th				ld 0x51234540
6 th	ld 0x5FFFFFF00			

After some time executing a particular path (which could be a path *different* from the paths in parts b.1-b.4) and with no further state changes caused by cosmic rays, we find that the final state of the caches is as follows.

Final State

Cache 0		
	Tag	MESI state
Set 0	0x5FFFFFFF	M
Set 1	0x5FFFFFFF	E
Set 2	0x5FFFFFFF	S
Set 3	0x5FFFFFFF	E

Cache 1		
	Tag	MESI state
Set 0	0x5FF000	I
Set 1	0x510000	S
Set 2	0x5FFFFFFF	S
Set 3	0x533333	I

Cache 2		
	Tag	MESI state
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFFFF	S
Set 3	0x533333	I

Cache 3		
	Tag	MESI state
Set 0	0x5FF000	M
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

- (c) [25 points] What is the *minimum* set of memory instructions that leads the system from the initial state to the final state? Indicate the set of instructions in order, and clearly specify the access type (ld/st), the address of each memory request, and the processor from which the request is generated.

7 Memory Consistency [65 points]

A programmer writes the following two C code segments. She wants to run them concurrently on a multicore processor, called SC, using two different threads, each of which will run on a different core. The processor implements *sequential consistency*, as we discussed in the lecture.

Thread T0		Thread T1	
Instr. T0.0	<code>a = X[0];</code>	Instr. T1.0	<code>Y[0] = 1;</code>
Instr. T0.1	<code>b = a + Y[0];</code>	Instr. T1.1	<code>*flag = 1;</code>
Instr. T0.2	<code>while(*flag == 0);</code>	Instr. T1.2	<code>X[1] *= 2;</code>
Instr. T0.3	<code>Y[0] += 1;</code>	Instr. T1.3	<code>a = 0;</code>

X, Y, and `flag` have been allocated in main memory, while `a` and `b` are contained in processor registers. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 0. Assume each line of the C code segment of a thread is a *single* instruction.

- (a) [15 points] What is the final value of `Y[0]` in the SC processor, after both threads finish execution? Explain your answer.

- (b) [15 points] What is the final value of `b` in the SC processor, after both threads finish execution? Explain your answer.

With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called RC, that implements *weak consistency*. As discussed in the lecture, the weak consistency model has no need to guarantee a strict order of memory operations. For this question, consider a very weak model where there is *no* guarantee on the ordering of instructions as seen by different cores.

- (c) [15 points] What is the final value of $Y[0]$ in the RC processor, after both threads finish execution? Explain your answer.

After several months spent debugging her code, the programmer learns that the new processor includes a `memory_fence()` instruction in its ISA. The semantics of `memory_fence()` is as follows for a given thread that executes it:

1. Wait (stall the processor) until *all* preceding memory operations from the thread complete in the memory system and become visible to other cores.
 2. Ensure *no* memory operation from any later instruction in the thread gets executed before the `memory_fence()` is retired.
- (d) [20 points] What *minimal* changes should the programmer make to the program above to ensure that the final value of `Y[0]` on RC is the same as that in part (a) on SC? Explain your answer.

8 Prefetching [45 points]

An ETH student writes two programs A and B and runs them on 3 different toy machines, M1, M2, and M3, to determine the type of the prefetching mechanism used in each of these 3 machines. She observes programs A and B to have the following access patterns to cache blocks. Note that the addresses are *cache block addresses*, not byte addresses.

Program A: 27 accesses

$a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,$
 $a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,$
 $a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64$

Program B: 501 accesses

$b, b + 2, b + 4, \dots, b + 998, b + 1000$

The student is able to measure the accuracy and coverage of the prefetching mechanism in each of the machines. The following table shows her measurement results:

	Machine M1		Machine M2		Machine M3	
	Coverage	Accuracy	Coverage	Accuracy	Coverage	Accuracy
Program A	6/27	6/27	0	0	1/3	9/26
Program B	499/501	499/501	0	0	499/501	499/500

The student knows the following facts about M1, M2, and M3 machines:

- The prefetcher prefetches into a fully-associative cache whose size is 8 cache blocks. The cache employs the FIFO (First-In First-Out) replacement policy.
- The prefetchers have large enough resources to detect and store access patterns.
- Each cache block access is separated long enough in time such that all prefetches issued can complete before the next access happens.
- There are 5 different possible choices for the prefetching mechanism:
 - 1) Markov prefetcher with a correlation table of 4 entries
 - 2) Markov prefetcher with a correlation table of 10 entries
 - 3) 1st-next-block prefetcher (degree = 1) – prefetches block $N + 1$ after seeing block N
 - 4) 4th-next-block prefetcher (degree = 1) – prefetches block $N + 4$ after seeing block N
 - 5) stride prefetcher
- None of the above-mentioned prefetchers employ confidence bits.
- The prefetchers start out with an empty table when each program A and B start execution.
- The prefetcher sends only one prefetch request after a program access (i.e., prefetch degree = 1).

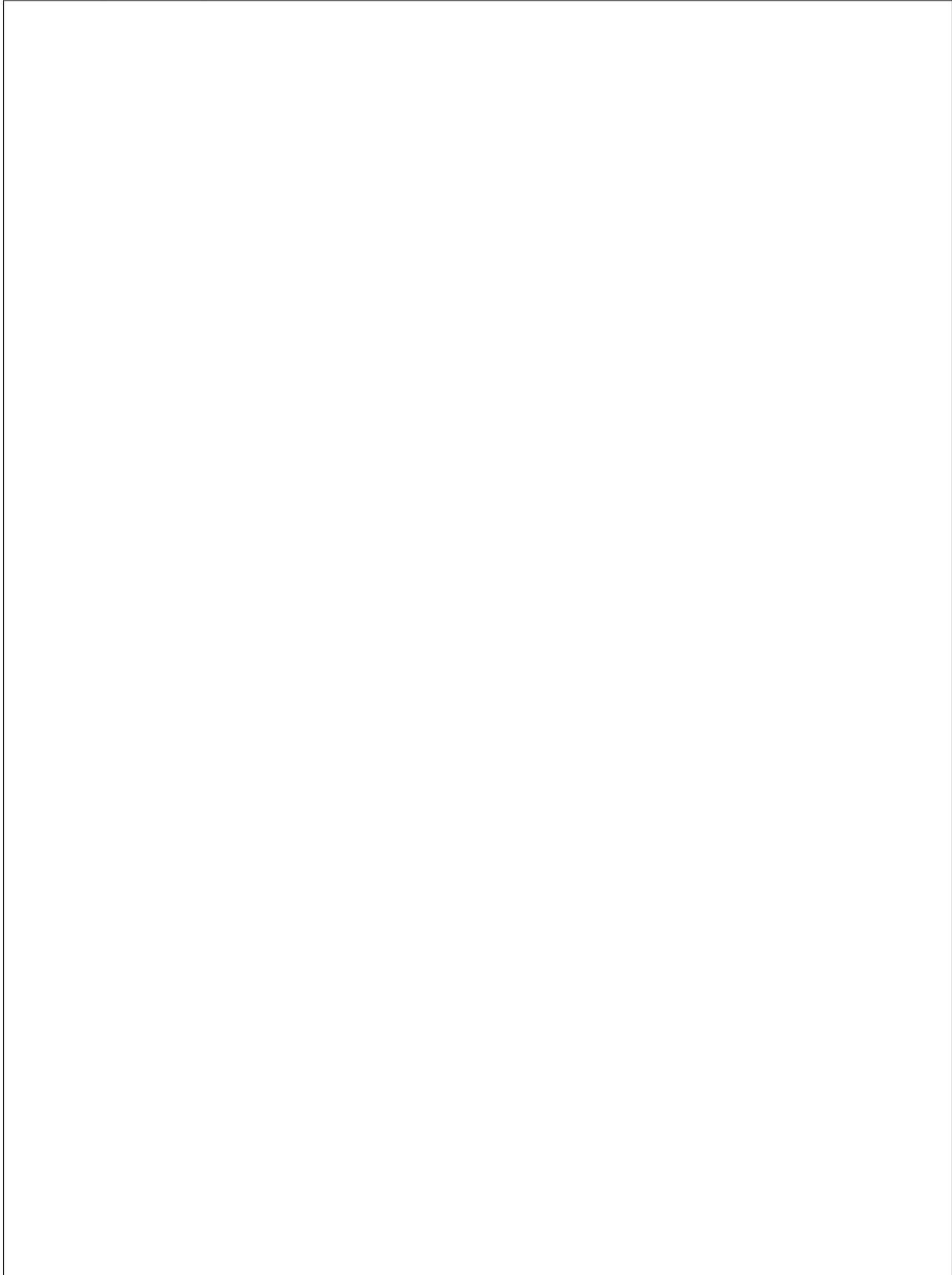
Determine what type of prefetching mechanism each of the above-mentioned machines use:

Machine M1:

Machine M2:

Machine M3:

Extra space for explanation:



9 BONUS: GPU Programming and Performance Analysis [75 points]

The following two program segments are executed on a GPU with C compute units. In each compute unit, one or more thread-blocks can run. Each thread-block is composed of threads that are grouped into warps of W threads.

In both programs, 2D thread-blocks are used. Each thread-block is identified by its block indices (bx, by) , and each thread is identified by its thread indices (tx, ty) . The size of a thread-block is $bdx * bdy$. Consider that a thread-block is decomposed into warps in a way that threads with consecutive tx and equal ty belong to the same warp. More specifically, the warp number for a thread (tx, ty) is $\frac{ty * bdx + tx}{\text{warp-size}}$.

The entire input size is $rows * cols$ integers. The size of an integer element is 4 bytes. The input is divided into tiles that are assigned to the thread-blocks.

`local_data` is an array in *local memory*, a fast on-chip memory that is used as a software-programmable cache. The amount of local memory per compute unit is S bytes. The threads of a thread-block can load data from *global memory* (i.e., the GPU off-chip memory) into local memory. The size of a global memory transaction is equal to the warp size times 4 bytes.

Program A:

```
__gpu_kernel_a(int* data, int rows, int cols){

    int* local_data[bdx * bdy];

    const int g_row = by * bdy + ty;
    const int g_col = bx * bdx + tx;
    const int l_row = ty;
    const int l_col = tx;
    const int pos    = g_row * cols + g_col;

    local_data[l_row * bdx + l_col] = data[pos];

    // Compute using local_data
}
```

Program B:

```
__gpu_kernel_b(int* data, int rows, int cols){

    int* local_data[bdx * bdy];

    const int g_row = bx * bdx + tx;
    const int g_col = by * bdy + ty;
    const int l_row = tx;
    const int l_col = ty;
    const int pos    = g_row * cols + g_col;

    local_data[l_row * bdy + l_col] = data[pos];

    // Compute using local_data
}
```

Please answer the questions on the next page.

- (a) [15 points] What is the maximum number of thread-blocks that run in *each* compute unit for programs A and B?

- (b) [15 points] Assuming that the GPU does *not* have caches, which program will execute faster? Why?

- (c) [15 points] Assume that the GPU has a single level of cache shared by all compute units. What will be the effect of this cache on the execution time of programs A and B?

- (d) [15 points] Assume that the access latency to the shared cache in part (c) is negligible. What should be the minimum size of the shared cache to guarantee that programs A and B have the same (or very similar) performance? (NOTE: The solution is independent of the warp scheduling policy).

- (e) [15 points] Now assume that *only one* thread-block is executed in each compute unit. Each thread-block in program A needs always T ms to complete its work, because the computation is very regular. What will be the total execution time of program A?

Initials: _____

Computer Architecture

January 29th, 2018

- SCRATCHPAD -

Initials: _____

Computer Architecture

January 29th, 2018

- SCRATCHPAD -

Initials: _____

Computer Architecture

January 29th, 2018

- SCRATCHPAD -