Family Name:	First Name:	Student ID:

Final Exam

Computer Architecture (263-2210-00L)

ETH Zürich, Fall 2018

Prof. Onur Mutlu

Problem 1 (40 Points):	Emerging Memory Technologies	
Problem 2 (70 Points):	Memory Scheduling	
Problem 3 (80 Points):	Asymmetric Multicore	
Problem 4 (55 Points):	Multicore Cache Partitioning	
Problem 5 (40 Points):	Cache Coherence	
Problem 6 (45 Points):	Memory Consistency	
Problem 7 (65 Points):	Processing-in-Memory	
Problem 8 (BONUS: 50 Points):	GPU Programming	
Total (445 (395 + 50 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. If you make assumptions, state your assumptions clearly and precisely.
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.**

This page intentionally left blank

1 Emerging Memory Technologies [40 points]

Researchers at Lindtel developed a new memory technology, L-RAM, which is non-volatile. The access latency of L-RAM is close to that of DRAM while it provides higher density compared to the latest DRAM technologies. L-RAM has one shortcoming, however: it has limited endurance, i.e., a memory cell stops functioning after 10^6 writes are performed to the cell (known as cell wear-out).

(a) [15 points] Lindtel markets a new computer system with L-RAM to have a lifetime of 2 years and the following specifications:

- 4 GBs of L-RAM as main memory with a *perfect* wear-leveling mechanism, i.e., writes are equally distributed over all the cells of L-RAM.
- The processor is in-order and there is no memory-level parallelism.
- It takes 4 ns to send a memory request from the processor to the memory controller and it takes 20 ns to send the request from the memory controller to L-RAM. The write latency of L-RAM is 40 ns.
- L-RAM is word-addressable. Thus, each write request writes 8 bytes to memory.

A student at ETH tests the lifetime of the system and finds that this new computer system *cannot* guarantee a lifetime of 2 years. She writes a program to wear out the entire L-RAM device as quickly as possible. How fast is she able to wear out the device? Show all work.

(b) [15 points] L-RAM works in the multi-level cell (MLC) mode in which each memory cell stores 2 bits. The student decides to improve the lifetime of L-RAM cells by using the single-level cell (SLC) mode. When L-RAM is used in SLC mode, the lifetime of each cell improves by a factor of 10 and the write latency decreases by 75%. What is the lifetime of the system using the SLC mode, if we repeat the experiment in part (a), with all else remaining the same in the system? Show your work.

(c) [10 points] Provide a mechanism that would increase the guaranteed lifetime of the computer system without changing the physical circuitry of L-RAM. From the baseline computer system in part (a), describe the changes required to guarantee a computer system lifetime of 2 years, with your mechanism. Be concrete and precise.

2 Memory Scheduling [70 points]

In lectures, we introduced a variety of ways to tackle memory interference. In this problem, we will look at the Blacklisting Memory Scheduler (BLISS) to reduce unfairness. There are two key aspects of BLISS that you need to know.

- When the memory controller services η consecutive requests from a particular application, this application is blacklisted. We name this non-negative integer η the **Blacklisting Threshold**.
- The blacklist is cleared periodically every **10000** cycles starting at $t = 0$.

To reduce unfairness, memory requests in BLISS are prioritized in the following order:

- Non-blacklisted applications' requests
- Row buffer hit requests
- Older requests

The memory system for this problem consists of 2 channels with 2 banks each. Tables 1 and 2 show the memory request stream in the same bank for both applications at different times ($t = 0$ and $t = 10$). For both tables, a request on the left-hand side is older than a request on the right-hand side in the same table. The applications do not generate more requests than those shown in Tables 1 and 2. The memory requests are labeled with numbers that represent the row position of the data within the accessed bank. Assume the following for all questions:

- A row buffer *hit* takes **100 cycles**.
- A row buffer *miss* (i.e., opening a row in a bank with a closed row buffer) takes **200 cycles**.
- A row buffer *conflict* (i.e., closing the currently open row and opening another one) takes **250 cycles**.
- All row buffers are closed at time $t = 0$

Application A (Channel 0, Bank 0)								
Application B (Channel 0, Bank 0)	Row 2	Row 2	Row 2	Row 2	Row 2	Row 3	Row 3	Row 4

Table 1: Memory requests of the two applications at $t = 0$

Application A (Channel 0, Bank 0)	Row 3	Row 7	Row 2	Row 0	Row 5	Row 3	Row 8	Row 9
Application B (Channel 0, Bank 0)	Row 2	Row 2	Row 2	Row 2	Row 2	Row 3	Row 3	Row 4

Table 2: Memory requests of the two applications at $t = 10$. Note that none of the Application B's existing requests are serviced yet.

(a) [15 points] Compute the slowdown of each application using the FR-FCFS scheduling policy after both threads ran to completion. We define:

$$slowdown = \frac{memory\ latency\ of\ the\ application\ when\ run\ together\ with\ other\ applications}{memory\ latency\ of\ the\ application\ when\ run\ alone}$$

Show your work.

(b) [15 points] If we use the BLISS scheduler, for what value(s) of η (the Blacklisting Threshold) will the slowdowns of **both** applications be equal to those obtained with FR-FCFS?

(c) [15 points] For what value(s) of η (the Blacklisting Threshold) will the slowdown of A be < 1.5?

(d) [15 points] For what value(s) of η (the Blacklisting Threshold) will B experience the maximum slowdown it can possibly experience with the Blacklisting Scheduler?

(e) [10 points] What is a simple mechanism (that we discussed in lectures) that we can use instead of BLISS to make the slowdowns of both A and B equal to 1.00?

3 Asymmetric Multicore [80 points]

A microprocessor manufacturer asks you to design an asymmetric multicore processor for modern workloads. You should optimize it assuming a workload with 80% of its work in the parallel portion. Your design contains one large core and several small cores, which share the same die. Assume the total die area is 32 units.

- *Large core*: For a large core that is n times faster than a single small core, you will need n^3 units of die area (n is a positive integer). The dynamic power of this core is $6 \times n$ Watts and the static power is n Watts.
- *Small cores*: You will fit as many small cores as possible, after placing the large core. A small core occupies 1 unit of die area. Its dynamic power is 1 Watt and its static power is 0.5 Watts.

The parallel portion executes *only* on the small cores, while the serial portion executes *only* on the large core.

Please answer the following questions. Show your work. Express your equations and solve them. You can approximate some computations, and get partial or full credit.

(a) [15 points] What configuration (i.e., number of small cores and size of the large core) results in the best performance?

(b) [10 points] The energy consumption should also be a metric of reference in your design. Compute the energy consumption for the best configuration in part (a).

(c) For the best configuration obtained in part (a), you are considering to use the large core to collaborate with the small cores on the execution of the parallel portion.

(i) [10 points] What is the overall performance improvement, compared to the performance obtained in part (a), if the large core collaborates on the parallel portion?

(ii) [10 points] What is the overall energy change, compared to the energy obtained in part (b), if the large core collaborates on the parallel portion?

(iii) [5 points] Discuss whether it is worth using the large core to collaborate with the small cores on the execution of the parallel portion.

(d) [15 points] Now assume that the serial portion can be optimized, i.e., the serial portion becomes smaller. This gives you the possibility of reducing the size of the large core, and still improving performance. For a large core with an area of $(n-1)^3$, where n is the value obtained in part (a), what should be the fraction of serial portion that would lead to better performance than in part (a)?

(e) [15 points] Your design is so successful for desktop processors that the company wants to produce a similar design for mobile devices. The power budget becomes a constraint. For a maximum of total power of 20W, how much would you need to reduce the dynamic power consumption of the large core, if at all, for the best configuration obtained in part (a)? Assume again that the parallel fraction is 80% of the workload. (Hint: Express the dynamic power of the large core as $D \times n$ Watts, where D is a constant).

4 Multicore Cache Partitioning [55 points]

Suppose we have a system with 32 cores that share a physical second-level cache. Assume each core is running a single single-threaded application, and all 32 cores are concurrently running applications. Assume that the page size of the architecture is 8KB, the block size of the cache is 128 bytes, and the cache uses LRU replacement. We would like to ensure each application gets a *dedicated* space in this shared cache without any interference from other cores. We would like to enforce this using the OS-based page coloring mechanism to partition the cache, as we discussed in lecture. Recall that with page coloring, the operating system ensures, using virtual memory mechanisms, that the applications do not contend for the same space in the cache.

(a) [10 points] What is the minimum size the L2 cache needs to be such that each application is allocated its dedicated space in the cache via page coloring? Show your work.

(b) [10 points] Assume the cache is 4MB, 32-way associative. Can the operating system ensure that the cache is partitioned such that no two applications interfere for cache space? Show your work.

(c) Assume you would like to design a 32MB shared cache such that the operating system has the ability to ensure that the cache is partitioned such that no two applications interfere for cache space.

  (i) [5 points] What is the minimum associativity of the cache such that this is possible? Show your work.

(ii) [10 points] What is the maximum associativity of the 32MB cache such that this is possible? Show your work.

(d) [5 points] Suppose we decide to change the cache design and use utility based cache partitioning (UCP) to partition the cache, instead of OS-based page coloring. Assume we would like to design a 4MB cache with a 128-byte block size. What is the minimum associativity of the cache such that each application is guaranteed a minimum amount of space without interference? Recall that UCP aims to minimize the cache miss rate by allocating more cache ways to applications that obtain the most benefit from more ways, as we discussed in lecture.

(e) [5 points] Is it desirable to implement UCP on a cache with this minimum associativity? Why, why not? Explain.

(f) [5 points] What is the maximum associativity of a 4MB cache that uses UCP such that each application is guaranteed a minimum amount of space without interference?

(g) [5 points] Is it desirable to implement UCP on a cache with this maximum associativity? Why, why not? Explain.

5 Cache Coherence [40 points]

We have a system with 4 byte-addressable processors {P0, P1, P2, P3}. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. All caches are connected to and actively snoop a global bus, and cache coherence is maintained using the MESI protocol, as we discussed in class. Note that on a write to a cache block in the S state, the block will transition directly to the M state. Accessible memory addresses range from `0x00000` − `0xfffff`.

Each processor executes the following instructions in a *sequentially consistent* manner:

P0	
0	`st r0, 0x1ff40`
-	
-	

P1	
1	`st r0, 0x110c0`
2	`st r1, 0x11080`
3	`ld r2, 0x1ff00`

P2	
4	`ld r0, 0x1ff40`
5	`ld r1, 0x110f0`
-	

P3	
-	
-	
-	

After executing the above 6 memory instructions, the *final* tag store state of each cache is as follows:

Final Tag Store States

Cache for P0		
	Tag	*MESI state*
Set 0	0x1ff	S
Set 1	0x1ff	S
Set 2	0x110	I
Set 3	0x110	I

Cache for P1		
	Tag	*MESI state*
Set 0	0x1ff	S
Set 1	0x1ff	I
Set 2	0x110	M
Set 3	0x110	M

Cache for P2		
	Tag	*MESI state*
Set 0	0x10f	I
Set 1	0x1ff	S
Set 2	0x10f	M
Set 3	0x110	I

Cache for P3		
	Tag	*MESI state*
Set 0	0x133	E
Set 1	0x000	I
Set 2	0x000	I
Set 3	0x10f	I

(a) [30 points] Fill in the following tables with the *initial* tag store states (i.e., *Tag* and *MESI* state) before having executed the six memory instructions shown above. Answer X if a tag value is unknown, and for the *MESI* states, write in *all possible values* (i.e., M, E, S, and/or I).

Initial Tag Store States

Cache for P0		
	Tag	*MESI state*
Set 0		
Set 1		
Set 2		
Set 3		

Cache for P1		
	Tag	*MESI state*
Set 0		
Set 1		
Set 2		
Set 3		

Cache for P2		
	Tag	*MESI state*
Set 0		
Set 1		
Set 2		
Set 3		

Cache for P3		
	Tag	*MESI state*
Set 0		
Set 1		
Set 2		
Set 3		

(b) [10 points] In what order did the memory operations enter the coherence bus?

time →					

6 Memory Consistency [45 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	`X[0] = 1;`	Instr. T1.0	`X[0] = 0;`
Instr. T0.1	`X[0] += 1;`	Instr. T1.1	`flag[0] = 1;`
Instr. T0.2	`while(flag[0] == 0);`	Instr. T1.2	`b = X[0];`
Instr. T0.3	`a = X[0];`		
Instr. T0.4	`X[0] = a * 2;`		

`X` 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) [10 points] What could be possible final values of `a` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

(b) [10 points] What could be possible final values of `X[0]` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

(c) [10 points] What could be possible final values of `b` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

(d) [15 points] The programmer wants `a` and `b` to have the same value at the end of the execution of both threads. The final value of `a` and `b` should be the same value as in the original program (i.e., the possible final values of `a` that you found in part (a)). What *minimal* changes should the programmer make to the program?
(Hint: You can use more flags if necessary.)

7 Processing-in-Memory [65 points]

You have been hired to accelerate ETH's student database. After profiling the system for a while, you found out that one of the most executed queries is to *"select the hometown of the students that are from Switzerland and speak German"*. The attributes *hometown*, *country*, and *language* are encoded using a four-byte binary representation. The database has 32768 (2^{15}) entries, and each attribute is stored contiguously in memory. The database management system executes the following query:

```
bool position_hometown[entries];
for(int i = 0; i < entries; i++){
    if(students.country[i] == "Switzerland" && students.language[i] == "German"){
        position_hometown[i] = true;
    }
    else{
        position_hometown[i] = false;
    }
}
```

(a) [25 points] You are running the above code on a single-core processor. Assume that:

- Your processor has an 8 MB direct-mapped cache, with a cache line of 64 bytes.
- A hit in this cache takes one cycle and a miss takes 100 cycles for both load and store operations.
- All load/store operations are serialized, i.e., the latency of multiple memory requests cannot be overlapped.
- The starting addresses of *students.country*, *students.language*, and *position_ hometown* are 0x05000000, 0x06000000, 0x07000000 respectively.
- The execution time of a non-memory instruction is zero (i.e., we ignore its execution time).

How many cycles are required to run the query? Show your work.

(b) Recall that in class we discussed AMBIT, which is a DRAM design that can greatly accelerate Bulk Bitwise Operations by providing the ability to perform bitwise AND/OR/XOR of two rows in a subarray. AMBIT works by issuing back-to-back ACTIVATE (A) and PRECHARGE (P) operations. For example, to compute AND, OR, and XOR operations, AMBIT issues the sequence of commands described in the table below (e.g., $AAP(X, Y)$ represents double row activation of rows X and Y followed by a precharge operation, $AAAP(X, Y, Z)$ represents triple row activation of rows X, Y, and Z followed by a precharge operation).

In those instructions, AMBIT copies the source rows D_i and D_j to auxiliary rows (B_i). Control rows C_i dictate which operation (AND/OR) AMBIT executes. The DRAM rows with dual-contact cells (i.e., rows DCC_i) are used to perform the bitwise NOT operation on the data stored in the row. Basically, copying a source row to DCC_i flips all bits in the source row and stores the result in both the source row and DCC_i. Assume that:

- The DRAM row size is **8 Kbytes**.
- An ACTIVATE command takes 50 cycles to execute.
- A PRECHARGE command takes 20 cycles to execute.
- DRAM has a single memory bank.
- The syntax of an AMBIT operation is: *bbop_* [and/or/xor] *destination, source_1, source_2.*
- Addresses 0x08000000 and 0x09000000 are used to store partial results.
- The rows at addresses 0x0A000000 and 0x0B00000 store the codes for *"Switzerland"* and *"German"*, respectively, in each four bytes throughout the entire row.

$D_k = D_i$ **AND** D_j	$D_k = D_i$ **OR** D_j	$D_k = D_i$ **XOR** D_j
		AAP (D_i, B_0)
		AAP (D_j, B_1)
		AAP (D_i, DCC_0)
AAP (D_i, B_0)	AAP (D_i, B_0)	AAP (D_j, DCC_1)
AAP (D_j, B_1)	AAP (D_j, B_1)	AAP (C_0, B_2)
AAP (C_0, B_2)	AAP (C_1, B_2)	AAAP (B_0, DCC_1, B_2)
AAAP (B_0, B_1, B_2)	AAAP (B_0, B_1, B_2)	AAP (C_0, B_2)
AAP B_0, D_k	AAP B_0, D_k	AAAP (B_1, DCC_0, B_2)
		AAP (C_1, B_2)
		AAAP (B_0, B_1, B_2)
		AAP (B_0, D_k)

i) [20 points] The following code aims to execute the query *"select the hometown of the students that are from Switzerland and speak German"* in terms of Boolean operations to make use of AMBIT. Fill in the blank boxes such that the algorithm produces the correct result. Show your work.

```
for(int i = 0; i < [          ] ; i++){

  bbop_[          ] 0x08000000, 0x05000000 + i*8192, 0x0A000000;

  bbop_[          ] 0x09000000, 0x06000000 + i*8192, 0x0B000000;

  bbop_[          ] 0x07000000, 0x08000000, 0x09000000;
}
```

ii) [20 points] How much speedup does AMBIT provide over the baseline processor when executing the same query? Show your work.

8 BONUS: GPU Programming [50 points]

An inexperienced CUDA programmer is trying to optimize her first GPU kernel for performance. After writing the first version of the kernel, she wants to find the best *execution configuration* (i.e., grid size and block size[1]).

As she assigns one thread per input element, calculating the grid size (i.e., total number of blocks) is trivial. For N input elements, the grid size is $\lceil \frac{N}{block\_size} \rceil$, where $block\_size$ is the number of threads per block. So, the challenging part will be to figure out what is the block size that produces the best performance. She will try 5 different block sizes (64, 128, 256, 512, and 1024 threads).

She has learned that a general recommendation for kernel optimization is to maximize the *occupancy* of the GPU cores, i.e., Streaming Multiprocessors (SMs). Occupancy is defined as the ratio of active threads to the maximum possible number of active threads per SM.

In order to calculate the occupancy, it is necessary to take the available SM resources into account. She knows that in each SM of her GPU:

- The total *scratchpad memory* or *shared memory* is 16 KB.
- The total *number of 4-byte registers* is 16384.

In her first version of the kernel code, each thread needs 2 4-byte elements in shared memory for its private use. In addition, each block needs 10 4-byte elements in shared memory for communication across threads.

She has also learned that she can obtain the number of registers that each thread needs by using a special compiler flag. This way, she finds that each thread in the first version of the kernel uses 9 registers.

(a) [15 points] After reasoning some time about the amount of shared memory that her code needs, she decides to first test a block size of 128 threads. Why do you think she chose that number? Show your work.

[1]We use NVIDIA terminology in this question.

However, after testing other block sizes, she finds out that using 256 threads per block provides higher performance than using 128. She does not understand why, so she looks for some information in the documentation of her GPU that can lead her to an explanation. There, she finds two more SM hardware constraints. In each SM:

- The *maximum number of blocks* is 8.
- The *maximum number of threads* is 2048.

Take into account these new constraints when answering parts (b), (c), and (d).

(b) [10 points] Can you explain why using 256 threads per block perform better than 128? Show your work.

(c) [15 points] What is the occupancy limitation due to register usage, if any? Explain and show your work.

(d) [10 points] The performance obtained by the first kernel version does not fulfill the acceleration needs. Thus, the programmer writes a second kernel version that reduces the number of instructions at the expense of using one more register per thread. What would be the highest occupancy for the second kernel? For what block size(s)?

- SCRATCHPAD -

- SCRATCHPAD -