Family Name: First Name: Student ID:

Midterm Exam

Computer Architecture (263-2210-00L)

ETH Zürich, Fall 2019

Prof. Onur Mutlu

Problem 1 (80 Points):	RowHammer	
Problem 2 (60 Points):	DRAM Refresh	
Problem 3 (60 Points):	VRT and DRAM Refresh	
Problem 4 (60 Points):	In-DRAM Bit Serial Computation	
Problem 5 (60 Points):	Genome Analysis	
Problem 6 (60 Points):	Low-Latency DRAM	
Total (380 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 RowHammer [80 points]

1.1 RowHammer Properties

Determine whether each of following statements is true or false. *Note: we will subtract 1 point for each* ***incorrect*** *answer. (The minimum score you can get for this question is 0 point.)*

(a) [2 points] Cells in a DRAM with a smaller technology node are more vulnerable to RowHammer.

`1. True` `2. False`

(b) [2 points] Cells which have shorter retention times are especially vulnerable to RowHammer.

`1. True` `2. False`

(c) [2 points] The vulnerability of cells in a victim row to RowHammer depends on the data stored in the victim row.

`1. True` `2. False`

(d) [2 points] The vulnerability of cells in a victim row to RowHammer depends on the data stored in the aggressor row.

`1. True` `2. False`

(e) [2 points] RowHammer-induced errors are mostly repeatable.

`1. True` `2. False`

1.2 RowHammer Attacks

In order to characterize the vulnerability of your DRAM device to RowHammer attacks, you must be able to induce RowHammer errors. Assume the following about the target system:

- The CPU has a single in-order processor, and does not implement virtual memory.
- The physical memory address is 16 bits.
- The DRAM subsystem consists of two channels, four banks per channel, and 64 rows per bank.
- The memory controller employs open-page policy.
- The DRAM modules and the memory controller do not employ any remapping or scrambling schemes for the physical address.
- All the cells in the DRAM subsystem are equally vulnerable to RowHammer-induced errors.

You implement codes based on instructions shown in Table 1.

Instruction	**Description**	**Functionality**
`B LABEL`	Unconditional Branch	`PC = LABEL`
`STORE IMM, Rs`	Store word to memory	`MEM[IMM] = Rs`
`CLFLUSH IMM`	Cache line flush	`Flush cache line containing IMM`

Table 1: Instruction Descriptions.

(a) [10 points] You run Code 1 below, but you cannot observe any errors in the target system. You figured out that the number of activations is much lower than your expectation. Give reason(s) as to why Code 1 cannot introduce a sufficient amount of activations.

Code 1

```
LOOP:
  STORE 0x8732, R0
  CLFLUSH 0x8732
  B LOOP
```

(b) [20 points] You try Codes 2a, 2b, and 2c, but find that *only one of them can induce RowHammer errors* in your DRAM subsystem. Which code segment is the one that can induce RowHammer errors? Justify your answer.

Code 2a

```
LOOP:
  STORE 0x8732, R0
  STORE 0x98CD, R1
  CLFLUSH 0x8732
  CLFLUSH 0x98CD
  B LOOP
```

Code 2b

```
LOOP:
  STORE 0xF1AB, R0
  STORE 0x0054, R1
  CLFLUSH 0xF1AB
  CLFLUSH 0x0054
  B LOOP
```

Code 2c

```
LOOP:
  STORE 0x2B97, R0
  STORE 0xDA68, R1
  CLFLUSH 0x2B97
  CLFLUSH 0xDA68
  B LOOP
```

1.3 RowHammer Mitigation Mechanisms

To identify a viable RowHammer mitigation mechanism for your system, you compare the two following mitigation mechanisms:

Mechanism A. The memory controller maintains a counter for every row, which increments every time the corresponding row is activated. If the counter value for a row exceeds a threshold value T, the memory controller activates the row's two adjacent rows and resets the counter.

Mechanism B. Each time a row is closed (or precharged), the memory controller flips a biased coin with a probability p of turning up heads, where $p << 1$. If the coin turns up heads, the memory controller activates one of its adjacent rows where either of the two adjacent rows are selected with equal probability ($p/2$).

(a) [5 points] You set T for Mechanism A to 164 K based on the value of the Maximum Activation Count (MAC, i.e., the maximum number of times a row can be activated without inducing RowHammer errors in its adjacent rows) reported by the DRAM manufacturer. Calculate the number of bits required for counters in a memory controller which manages a single channel, 2 ranks per channel, 8 banks per rank, and 2^{15} rows per bank.

(b) [5 points] How does the answer to (a) change when both the number of rows per bank and the number of banks per chip are doubled?

(c) [10 points] You profile the memory access pattern of the target system, and observe that the same pattern repeats exactly every 64 ms (the current refresh interval). Table 2 shows the number of activations for each row within a 64-ms time interval in a descending order. Given values $T = 164$ K for Mechanism A and $p = 0.001$ for Mechanism B, calculate the expected number of additional activations within a 64-ms time interval under each technique.

Row Index	# of ACTs
0x7332F	73 K
0x1802C	64 K
0x03F05	32 K
0x5FF02	10 K
...	...
Total	480 K

Table 2: Number of Activations for Each Row.

(d) [5 points] How does the answer to (c) change when both the number of rows per bank and the number of banks per chip are doubled? Assume that the memory access pattern does not change.

(e) [10 points] What is the *common challenge* to implement the above mechanisms in the commodity systems?

(f) [5 points] How can you address the common challenge?

2 DRAM Refresh [60 points]

2.1 Basics [15 points]

A memory system is composed of eight banks, and each bank contains 2^{16} rows. Every DRAM row refresh is initiated by a command from the memory controller, and it refreshes a single row in a single DRAM bank. Each refresh command keeps the command bus busy for 5 ns. We define *command bus utilization* as the fraction of total execution time during which the command bus is occupied.

1. [5 points] Given that the refresh interval is 64ms, calculate the command bus utilization of refresh commands. Show your work step-by-step.

2. [10 points] If 70% of all rows can withstand a refresh interval of 256 ms, how does the command bus utilization of refresh commands change? Calculate the reduction $(1 - \frac{new}{old})$ in bus utilization. Show your work step-by-step.

2.2 VRL: Variable Refresh Latency [45 points]

In this question, you are asked to evaluate "Variable Refresh Latency," proposed by Das et al. in DAC 2018.[1]

The paper presents two key observations:

- First, a cell's charge reaches 95% of the maximum charge level in 60% of the nominal latency value during a refresh operation. In other words, the last 40% of the refresh latency is spent to increase the charge of a cell from 95% to 100%. Based on this observation, the paper defines two types of refresh operations: (1) *full refresh* and (2) *partial refresh*. Full refresh uses the nominal latency value and restores the cell charge to 100%, while the latency of partial refresh is only 60% of the nominal latency value and it restores 95% of the charge.

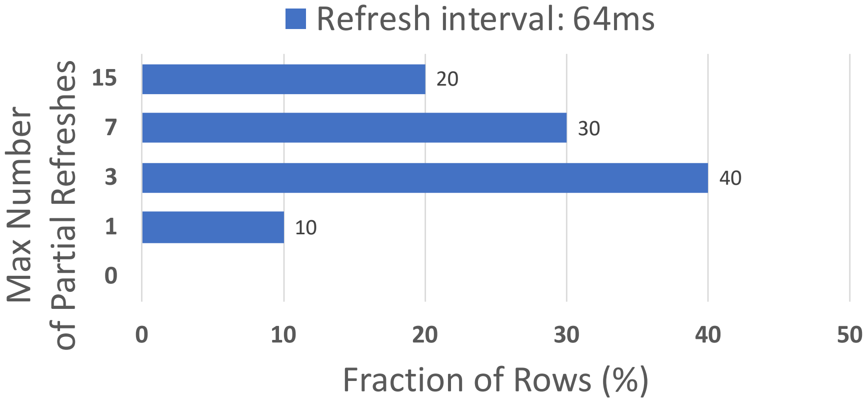
[1] Das, A. et al., *"VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency."* In Proceedings of the 55th Annual Design Automation Conference (DAC), 2018.

- Second, a fully refreshed cell operates correctly even after multiple partial refreshes, but it needs to be fully refreshed again after a finite number of partial refreshes. The maximum number of partial refreshes before a full refresh is required varies from cell to cell.

The **key idea** of the paper is to apply a *full refresh* operation **only when necessary** and use *partial refresh* operations at all other times.

(a) [15 points] Consider a case in which:

- Each row must be refreshed every 64 ms. In other words, the refresh interval is 64 ms.
- Row refresh commands are evenly distributed across the refresh interval. In other words, all rows are refreshed exactly once in any given 64 ms time window.
- You are given the following plot, which shows *the distribution of the maximum number of partial refreshes* across all rows of a particular bank. For example, if the maximum number of refreshes is three, those rows can be partially refreshed for at most three refresh intervals, and the fourth refresh operation must be a full refresh.
- If all rows were always fully refreshed, the time that a bank is busy serving the refresh requests within a refresh interval would be T.



How much time does it take (in terms of T) for a bank to refresh all rows within a refresh interval, after applying Variable Refresh Latency?

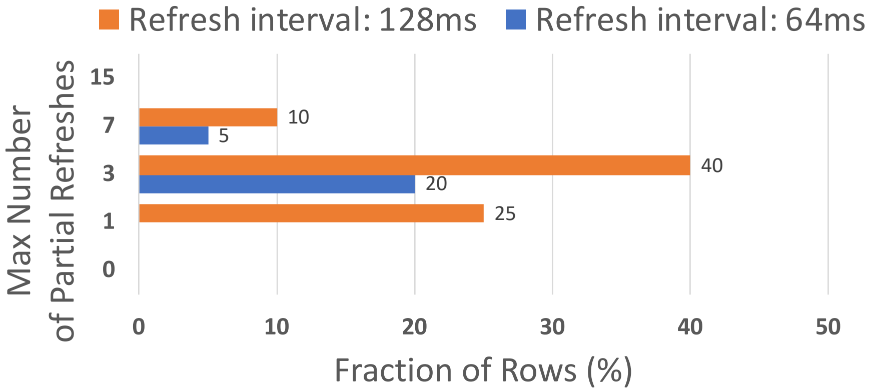
(b) [15 points] You find out that you can relax the refresh interval, and define your baseline as follows:

- 75% of the rows are refreshed at every 128ms; 25% of the rows are refreshed at every 64ms.
- Refresh commands are evenly distributed in time.
- All rows are always fully refreshed.
- A single refresh command costs $0.2/N\ ms$, where N is the number of rows in a bank.
- *Refresh overhead* is defined as the fraction of time that a bank is busy, serving the refresh requests over a very large period of time.

Calculate the refresh overhead for the baseline.

(c) [15 points] Consider a case where:

- 75% of the rows are refreshed at every 128ms; 25% of the rows are refreshed at every 64ms.
- Refresh commands are evenly distributed in time.
- You are given the following plot, which shows *the distribution of the maximum number of partial refreshes* across all rows of a particular bank.
- A single refresh command costs $0.2/N\ ms$, where N is the number of rows in a bank.
- *Refresh overhead* is defined as the fraction of time that a bank is busy, serving the refresh requests over a very large period of time.



Calculate the refresh overhead. Show your work step-by-step. Then, compare it against the baseline configuration (the previous question). How much reduction ($1 - \frac{new}{old}$) do you see in the performance overhead of refreshes?

3 VRT and DRAM Refresh [60 points]

You observe that your system suffers from random bit flips in the main memory. All the observations suggest that the source of these bit errors is likely to be Variable Retention Time (VRT). Unfortunately, your hardware does not implement any ECC mechanism to correct those bit flips. You need a quick fix to mitigate these bit errors. Answer the questions for the following system configuration.

- The memory controller refreshes every DRAM row at every 64ms.
- The memory subsystem consists of one channel, one rank, and 16 banks.
- The total capacity of DRAM is 2GB. Each DRAM row contains 4kB. The cache line size is 64B.
- A bank spends 0.5ms busy refreshing rows in a 64ms time window.
- A DRAM cell that is vulnerable to VRT is called VRT cell regardless of its leakage state.
- VRT cells are uniformly distributed across the main memory.
- A VRT cell can retain its data for 128ms and 16ms in low- and high-leakage states, respectively.

(a) [15 points] Evaluate the idea of increasing the refresh rate. To avoid having VRT-related bit errors, refresh rate should support the worst retention time.

What should be the new refresh rate? How much time would refresh operations keep a bank busy in a 64ms of time window? Is increasing refresh rate a viable solution?

(b) [15 points] Your company also bought the newer generation of the same DRAM model with larger storage capacity that implements the same channel/rank/bank organization. Due to the higher capacity in the same chip area, the new generation employs smaller cells, which may have smaller retention time. Assume that:

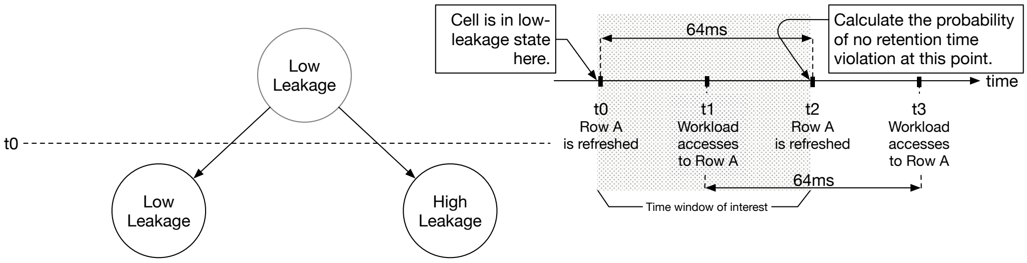
- The retention time of a DRAM cell is halved for both low- and high-leakage states (i.e., 64ms in low-leakage and 8ms in high-leakage),
- The latency of refreshing *one row* remains the same.
- The capacity of new DRAM model is 64GB.

What should be the new refresh rate? How much time would refresh operations keep a bank busy in a 64ms of time window? Is increasing refresh rate a viable solution?

(c) [30 points] Since this system is dedicated to run a particular workload, you need to analyze the workload access pattern to assess opportunities of having a low-cost solution. As the first step, you will consider **a single VRT cell**, which is located in Row A. Note that:

- The workload accesses to Row A exactly every 64ms, as shown at timestamps t1 and t3 below.
- The memory controller refreshes Row A every 64ms, as shown at timestamps t0 and t2 below.
- Refresh operations are not synchronized with the workload accesses. Assume that t2 can happen at any timestamp between t1 and t3 with equal probability.
- The cell changes its leakage state with a 30% of probability when the wordline is enabled.
- The cell initializes at low-leakage state, and gets immediately refreshed, as shown at timestamp t0 below.
- The cell retains its data for 128ms in low-leakage state and for 16ms in high-leakage state.

Calculate the **probability of not violating the retention time of the cell for 64ms of execution**. (Hint: You can create a tree for the probabilistic state transitions, as we partially provide below.)



Cell is in low-leakage state here.

Calculate the probability of no retention time violation at this point.

4 In-DRAM Bit Serial Computation [60 points]

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 of two rows in a subarray and NOT of one row. Since Ambit is logically complete, it is possible to implement any other logic gate (e.g., XOR). To be able to implement arithmetic operations, bit shifting is also necessary. There is no way of shifting bits in DRAM with a conventional layout, but it can be done with a bit-serial layout, as Figure 1 shows. With such a layout, it is possible to perform bit-serial arithmetic computations in Ambit.

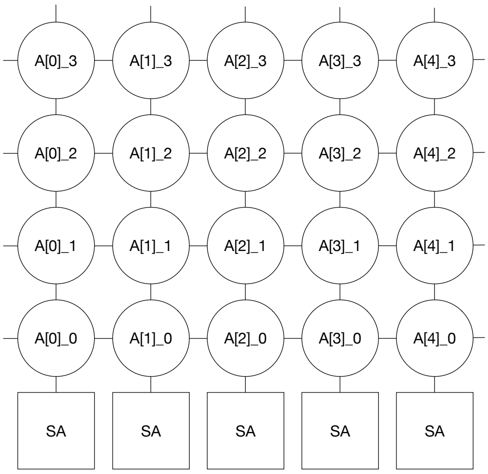


Figure 1: In-DRAM bit-serial layout for array `A`, which contains five 4-bit elements. DRAM cells in the same bitline contain the bits of an array element: `A[i]_j` represents bit `j` of element `i`.

We want to evaluate the potential performance benefits of using Ambit for arithmetic computations by implementing a simple workload, the element-wise addition of two arrays. Listing 1 shows a sequential code for the addition of two input arrays `A` and `B` into output array `C`.

Listing 1: Sequential CPU implementation of element-wise addition of arrays `A` and `B`.

```
for(int i = 0; i < num_elements; i++){
    C[i] = A[i] + B[i];
}
```

We compare two possible implementations of the element-wise addition of two arrays: a CPU-based and an Ambit-based implementation. We make two assumptions. First, we use the most favorable layout for each implementation (i.e., conventional layout for CPU, and bit-serial layout for Ambit). Second, both implementations can operate on array elements of any size (i.e., bits/element):

- *CPU-based implementation*: This implementation reads elements of `A` and `B` from memory, adds them, and writes the resulting elements of `C` into memory.

  Since the computation is simple and regular, we can use a simple analytical performance model for the execution time of the CPU-based implementation: $t_{cpu} = K \times num\_operations + \frac{num\_bytes}{M}$, where K represents the cost per arithmetic operation and M is the DRAM bandwidth.

- *Ambit-based implementation*: This implementation assumes a bit serial layout for arrays `A`, `B`, and `C`. It performs additions in a bit serial manner, which only requires XOR, AND, and OR operations, as you can see in the 1-bit full adder in Figure 2.

  Ambit implements these operations 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 Table 3, where $AAP(X, Y)$ represents double row activation of rows X and Y followed by a precharge operation, and $AAAP(X, Y, Z)$ represents triple row activation of rows X, Y, and Z followed by a precharge operation.

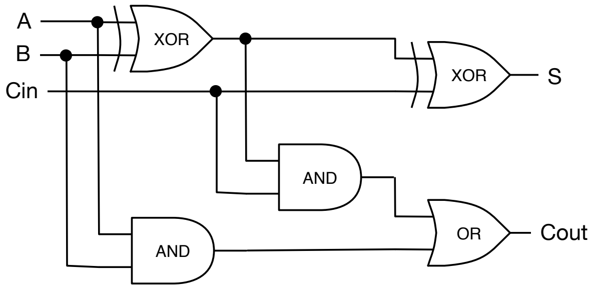


Figure 2: 1-bit full adder.

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, the NOT operation copies a source row to DCC_i, flips all bits of the 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 20ns to execute.
- A PRECHARGE command takes 10ns to execute.
- DRAM has a single memory bank.
- The syntax of an Ambit operation is: *bbop_* [and/or/xor] *destination, source_1, source_2.*
- The rows at addresses 0x00700000, 0x00800000, and 0x00900000 are used to store partial results. Initially, they contain all zeroes.
- The rows at addresses 0x00A00000, 0x00B00000, and 0x00C00000 store arrays `A`, `B`, and `C`, respectively.
- These are all byte addresses. All these rows belong to the same DRAM subarray.

Table 3: Sequences of ACTIVATE and PRECHARGE operations for the execution of Ambit's AND, OR, and XOR.

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

(a) [10 points] For the CPU-based implementation, you want to obtain K and M. To this end, you run two experiments. In the first experiment, you run your CPU code for the element-wise array addition for 65,536 4-bit elements and measure $t_{cpu} = 100$ us. In the second experiment, you run the STREAM-Copy benchmark for 102,400 4-bit elements and measure $t_{cpu} = 10$ us. The STREAM-Copy benchmark simply copies the contents of one input array `A` to an output array `B`. What are the values of K and M?

(b) [20 points] Write the code for the Ambit-based implementation of the element-wise addition of arrays `A` and `B`. The resulting array is `C`.

(c) [20 points] Compute the maximum throughput (in arithmetic operations per second, OPS) of the Ambit-based implementation as a function of the element size (i.e., bits/element).

(d) [10 points] Determine the element size (in bits) for which the CPU-based implementation is faster than the Ambit-based implementation (Note: Use the same array size as in the previous part).

5 Genome Analysis [60 points]

During a process called read mapping in genome analysis, each read (i.e., genomic subsequence) is mapped to one or more locations in the reference genome based on the similarity between the read and the reference genome segment at that location. Potential mapping locations are identified based on the presence of exact short segments (i.e., *k-mers* where *k* is the length of the short segment) from the read sequence, in the reference genome. The locations of the k-mers in the reference genome are usually determined using a *hash table*. Each entry of the hash table stores a *key-value* pair, where the key is a k-mer and the value is a *list of locations* at which the k-mer occurs in the reference genome.

A challenge in designing such a hash table is deciding which k-mers to use as keys, as it affects the size of the hash table and the number of potential mapping locations, which affect the execution time of read mapping. In this question, you will be exploring the trade-offs between two strategies of k-mer selection:

(1) **Non-overlapping 4-mers**: Every *non-overlapping 4-mers* in the reference genome is used as a key in the hash table. For example, the reference AAAATTCA contains only two non-overlapping 4-mers: AAAA and TTCA. Thus, the hash table would have the following entries: {AAAA} → {1} and {TTCA} → {5}, where 1 and 5 are the start locations of the non-overlapping 4-mers (keys) in the reference.

(2) **Non-overlapping 4-mer minimizers**: For every non-overlapping 4-mer in the reference genome, the lexicographically minimum 4-mer of it and the two subsequent non-overlapping 4-mers is used as a key in the hash table. For example, the segment AAAATTCAACGGGCAG contains only two non-overlapping 4-mer minimizers, AAAA and ACGG. This is because AAAA is the lexicographically minimum k-mer among the first three consecutive k-mers (i.e., AAAA, TTCA, ACGG), and ACGG is the lexicographically minimum k-mer among the next three consecutive k-mers (i.e., TTCA, ACGG, and GCAG). Thus, the hash table would have the following entries: {AAAA} → {1} and {ACGG} → {9}, where 1 and 9 are the start locations of the minimizers (keys) in the reference.

Suppose that you would like to map a set of reads to the following reference genome. Note that the 4-mers are separated by '_' *only* to help you identify the 4-mers easily, so you should **not** count them when creating a list of locations for a key.

AAAA_ATAC_TGAT_CCTT_ATAC_GTTG_TAAG_GTTT_CAAA_GTTG_ATAC_TAAG_TGAT

Answer the following questions based on the information given above:

(a) [10 points] Please list all {key} → {value} entries in the hash table if we use all **non-overlapping 4-mers** as keys? The order of the entries is **not** important.

(b) [10 points] Please list all {key} → {value} entries in the hash table if we use all **non-overlapping 4-mer minimizers** as keys? Please list all the entries of this hash table. The order of the entries is **not** important.

(c) [20 points] Assume that we calculate the size of the hash table allocated in memory as: $2^{\lceil \log_2 e \rceil} + p$ bytes, where e is the total number of hash table entries and p is the total number of locations stored across all values. Calculate the memory footprint (in bytes) of each of the two hash tables you designed in (a) and (b). Show your work.

(d) [20 points] Now assume we can query the hash table in $\log_2 e$ cycles, where e is the total number of entries in the hash table. A read mapper queries the hash table using the *first 4-mer of the read* and calculates the *edit distance* between the read and the reference segment at *each* location returned by the hash table. Calculating the edit distance takes l^2 cycles where l is the length of the read. If the edit distance between the read and a segment in the reference is higher than a certain threshold, the read mapper discards the location. We refer to cycles spent calculating the edit distance for segments at discarded locations as wasted cycles. When we profile read mapping with *non-overlapping 4-mers* and *non-overlapping 4-mer minimizers* strategies, we find the $\frac{\text{wasted cycles}}{\text{total cycles}}$ ratios to be 0.9 and 0.8, respectively. Assume that we want to align the read: GTTGACCAATGA to the reference genome above. What are the *wasted cycles* when aligning the read using 1) *non-overlapping 4-mers* and 2) *non-overlapping 4-mer minimizers* strategies? Please show your work.

6 Low-Latency DRAM [60 points]

In class, we have seen the idea of Tiered-Latency DRAM (TL-DRAM). Recall that in TL-DRAM, each bitline of a subarray is segmented into two portions by adding isolation transistors in between, creating two segments on the bitline: the *near segment* and the *far segment*. The near segment is close to the sense amplifiers whereas the far segment is far away.

(a) [5 points] Why is accessing a row in the near segment faster in TL-DRAM compared to a commodity DRAM chip?

(b) [5 points] Why is accessing a row in the far segment slower in TL-DRAM compared to a commodity DRAM chip?

Now, assume that:

- We have a system that uses the near segment as a cache to the far segment, and the far segment contains main memory locations.
- The near segment is not visible to software and the rows that are cached in it are completely managed by the memory controller.
- The far segment is inclusive of the near segment.
- In each subarray, the far segment contains 496 rows whereas the total number of rows in the subarray is 512.

(c) [5 points] What is the capacity loss in main memory size when we use TL-DRAM as opposed to commodity DRAM with the same number of total DRAM rows? Express this as a fraction of total memory capacity lost (no need to simplify the fraction).

(d) [10 points] What is the tag store size that needs to be maintained on a per subarray basis in the DRAM controller if the near segment is used as a fully-associative write-back cache? Assume the replacement algorithm is *Most Recently Used* (MRU) and use the **minimum number of bits** possible to implement the replacement algorithm. Show your work.

Now assume near segment and far segment are *exclusive*. In other words, both contain memory rows, and a memory row can only be in one of the segments. When a memory row in the far segment is referenced, it is brought into the near segment by exchanging the MRU row in the near segment with the row in the far segment. Note that a row can end up in a *different* location in the far segment after being moved to the near segment and then back to the far segment.

(e) [5 points] When the near segment is used as an *exclusive* cache to the far segment, what is the capacity loss in main memory size when we use TL-DRAM as opposed to commodity DRAM with the same number of total DRAM rows? Express this as a fraction of total memory capacity lost (no need to simplify the fraction).

(f) [10 points] What is the tag store size that needs to be maintained on a per subarray basis in the DRAM controller if the near segment is used as an *exclusive* fully-associative write-back cache? Assume the replacement algorithm is MRU and use the **minimum number of bits** possible to implement the replacement algorithm. Show your work.

(g) [20 points] Assume the near and far segments are visible to the operating system (OS) and the OS can allocate physical pages in either of the segments. Answer the following questions as *True* or *False* and provide explanation to your answer.

- The OS *cannot* manage the near segment as a cache at granularity smaller than the physical page granularity.

- If the near segment is used as an OS-managed cache to store frequently-accessed pages, the cache can *only* be exclusive.

- There is zero memory capacity loss when the near segment is used as OS-managed cache.

- An OS-managed near segment cache does *not* incur *any* tag store overhead in the memory controller.

- SCRATCHPAD -

- SCRATCHPAD -

- SCRATCHPAD -

- SCRATCHPAD -