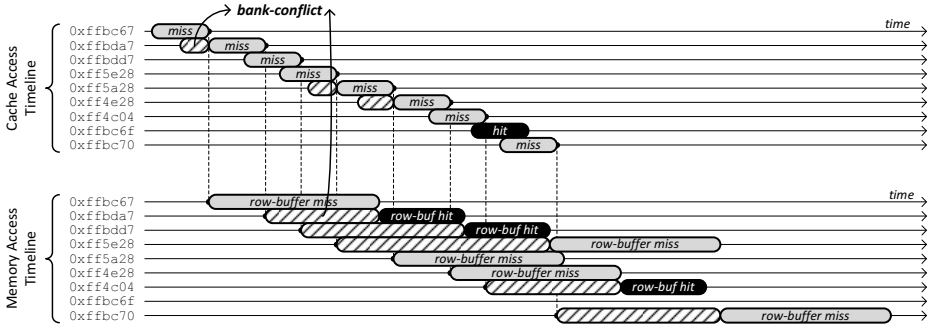
CMU 18-447 Introduction to Computer Architecture, Spring 2015
HW 6: Main Memory, Latency Tolerance, Prefetching and Runahead Execution

Instructor: Prof. Onur Mutlu
TAs: Rachata Ausavarungnirun, Kevin Chang, Albert Cho, Jeremie Kim, Clement Loh

Assigned: Wed., 4/1, 2015
Due: **Wed., 4/10, 2015 (Midnight)**
Handin: `/afs/ece/class/ece447/handin/hw6`
Please submit as ONE PDF: [andrewID].pdf

1 Banks [30 points]

A processor's memory hierarchy consists of a small SRAM L1-cache and a large DRAM main memory, both of which are banked. The processor has a 24-bit physical address space and does not support virtual memory (i.e., all addresses are physical addresses). An application has just started running on this processor. The following figure shows the timeline of memory references made by that application and how they are served in the L1-cache or main memory.



For example, the first memory reference made by the application is to byte-address `0xffbc67` (assume that all references are byte-sized reads to byte-addresses). However, the memory reference misses in the L1-cache (assume that the L1-cache is intially empty). Immediately afterwards, the application accesses main memory, where it experiences a row-buffer miss (initially, assume that all banks in main memory each have a row opened that will never be accessed by the application). Eventually, the cache block (and only that cache block) that contains the byte-address `0xffbc67` is fetched from memory into the cache.

Subsequent memory references may experience *bank-conflicts* in the L1-cache and/or main memory, if there is a previous reference still being served at that particular bank. Bank-conflicts are denoted as hatched shapes in the timeline.

The following table shows the address of the memory references made by the application in both hexadecimal and binary representations.

(a) Memory address table

Hexadecimal	Binary
ffbc67	1111 1111 1011 1100 0110 0111
ffbda7	1111 1111 1011 1101 1010 0111
ffbdd7	1111 1111 1011 1101 1101 0111
ff5e28	1111 1111 0101 1110 0010 1000
ff5a28	1111 1111 0101 1010 0010 1000
ff4e28	1111 1111 0100 1110 0010 1000
ff4c04	1111 1111 0100 1100 0000 0100
ffbc6f	1111 1111 1011 1100 0110 1111
ffbc70	1111 1111 1011 1100 0111 0000

From the above timelines and the table, your job is to answer questions about the processor's cache and main memory organization. Here are some assumptions to help you along the way.

- Assumptions about the L1-cache
  - Block size: ? (Power of two, greater than two)
  - Associativity: ? (Power of two, greater than two)
  - Total data-store size: ? (Power of two, greater than two)
  - Number of banks: ? (Power of two, greater than two)
  - Initially empty
- Assumptions about main memory
  - Number of channels: 1
  - Number of ranks per channel: 1
  - Number of banks per rank: ? (Power of two, greater than two)
  - Number of rows per bank: ? (Power of two, greater than two)
  - Number of cache-blocks per row: ? (Power of two, greater than two)
  - Contains the entire working set of the application
  - Initially, all banks have their 0^{th} row open, which is never accessed by the application

1.1 First, let's cover the basics

(a) Caches and main memory are sub-divided into multiple banks in order to allow parallel access. What is an alternative way of allowing parallel access?

Solution:
Multiporting, duplicating

(b) A cache that allows multiple cache misses to be outstanding to main memory at the same time is called what? (Two words or less. Hint: It's an adjective.)

Solution:
Non-blocking (or lockup-free)

(c) While cache misses are outstanding to main memory, what is the structure that keeps bookkeeping information about the outstanding cache misses? This structure often augments the cache.

Solution:
Miss status handling registers (MSHRs)

(d) Which is larger, an SRAM cell or a DRAM cell?

Solution:
SRAM cell

(e) What is the number of transistors and/or capacitors needed to implement each cell, including access transistor(s)?

Solution:
SRAM: 6T
DRAM: 1T-1C

1.2 Cache and memory organization

NOTE: For the following questions, assume that all offsets and indexes come from contiguous address bits.

(a) What is the L1-cache's block size in bytes? Which bit positions in the 24-bit physical address correspond to the cache block offset? (The least-significant bit in the physical address has a bit position of 0.)

Solution:
Block size: 16 bytes
Bit positions of block offset: 0-3

(b) How many banks are there in the L1-cache? Which bit positions in the 24-bit physical address correspond to the L1-cache bank index? (The least-significant bit in the physical address has a bit position of 0.)

Solution:
Number of L1-cache banks: 4
Bit positions of L1-cache bank index: 4-5

(c) How many banks are there in main memory? Which bit positions in the 24-bit physical address correspond to the main memory bank index? (The least-significant bit in the physical address has a bit position of 0.)

Solution:
Number of main memory banks: 8
Bit positions of main memory bank index: 10-12

(d) What kind of interleaving is used to map physical addresses to main memory?

Solution:
Row-interleaving

(e) To fully support a 24-bit physical address space, how many rows must each main memory bank have? Which bit positions in the 24-bit physical address correspond to the main memory row index? (The least-significant bit in the physical address has a bit position of 0.)

Solution:
Number of rows per main memory bank: 2048
Bit positions of row index: 13-23

(f) Each cache block within a row is called a *column*. How many columns are there in a single row? Which bit positions in the 24-bit physical address correspond to the main memory column index? (The least-significant bit in the physical address has a bit position of 0.)

Solution:
Number of columns per row: 64
Bit positions of column index: 4-9

2 Refresh [40 points]

A memory system has four channels, and each channel has two ranks of DRAM chips. Each memory channel is controlled by a separate memory controller. Each rank of DRAM contains eight banks. A bank contains 32K rows. Each row in one bank is 8KB. The minimum retention time among all DRAM rows in the system is 64 ms. In order to ensure that no data is lost, every DRAM row is refreshed once per 64 ms. Every DRAM row refresh is initiated by a command from the memory controller which occupies the command bus on the associated memory channel for 5 ns and the associated bank for 40 ns. Let us consider a 1.024 second span of time.

We define *utilization* (of a resource such as a bus or a memory bank) as the fraction of total time for which a resource is occupied by a refresh command.

For each calculation in this section, you may leave your answer in *simplified* form in terms of powers of 2 and powers of 10.

(a) How many refreshes are performed by the memory controllers during the 1.024 second period in total across all four memory channels?

2^{23} refreshes per channel; 2^{25} = 32M across 4 channels.

(b) What command bus utilization, across all memory channels, is directly caused by DRAM refreshes?

4.096

(c) What data bus utilization, across all memory channels, is directly caused by DRAM refreshes?

(d) What bank utilization (on average across all banks) is directly caused by DRAM refreshes?

2.048 = (32K * 16 * 40ns / 1.024s)

(e) The system designer wishes to reduce the overhead of DRAM refreshes in order to improve system performance and reduce the energy spent in DRAM. A key observation is that not all rows in the DRAM chips need to be refreshed every 64 ms. In fact, rows need to be refreshed only at the following intervals in this particular system:

Required Refresh Rate	Number of Rows
64 ms	2^5
128 ms	2^9
256 ms	all other rows

Given this distribution, if all rows are refreshed only as frequently as required to maintain their data, how many refreshes are performed by the memory controllers during the 1.024 second period in total across all four memory channels?

What command bus utilization (as a fraction of total time) is caused by DRAM refreshes in this case?

1.0243 = 5e-9 * (2**5 * 16 + 2**9 * 8 + (2**21 - 2**9 - 2**5) * 4) / (4*1.024)

(f) What DRAM data bus utilization is caused by DRAM refreshes in this case?

0

(g) What bank utilization (on average across all banks) is caused by DRAM refreshes in this case?

0.5121 = 40e-9 * (2**5 * 16 + 2**9 * 8 + (2**21 - 2**9 - 2**5) * 4) / (64*1.024)

(h) The system designer wants to achieve this reduction in refresh overhead by refreshing rows less frequently when they need less frequent refreshes. In order to implement this improvement, the system needs to track every row's required refresh rate. What is the minimum number of bits of storage required to track this information?

4 Mbit (2 bits per row)

(i) Assume that the system designer implements an approximate mechanism to reduce refresh rate using Bloom filters, as we discussed in class. One Bloom filter is used to represent the set of all rows which require a 64 ms refresh rate, and another Bloom filter is used to track rows which require a 128 ms refresh rate. The system designer modifies the memory controller's refresh logic so that on every potential refresh of a row (every 64 ms), it probes both Bloom filters. If either of the Bloom filter probes results in a "hit" for the row address, and if the row has not been refreshed in the most recent length of time for the refresh rate associated with that Bloom filter, then the row is refreshed. (If a row address hits in both Bloom filters, the more frequent refresh rate wins.) Any row that does not hit in either Bloom filter is refreshed at the default rate of once per 256 ms.

The false-positive rates for the two Bloom filters are as follows:

Refresh Rate Bin	False Positive Rate
64 ms	2^{-20}
128 ms	2^{-8}

The distribution of required row refresh rates specified in part (e) still applies.

How many refreshes are performed by the memory controllers during the 1.024 second period in total across all four memory channels?

false positives: 8192 + 2 (8194)

What command bus utilization results from this refresh scheme?

5e-9 * ((2**5 + 2) * 16 + (2**9 + 2**13) * 8 + (2**21 - 2**9 - 2**5 - 2 - 2**13) * 4) / (4*1.024)

What data bus utilization results from this refresh scheme?

0%

What bank utilization (on average across all banks) results from this refresh scheme?

0.5141 = 40e-9 * ((2**5 + 2) * 16 + (2**9 + 2**13) * 8 + (2**21 - 2**9 - 2**5 - 2 - 2**13) * 4) / (64*1.024)

3 Prefetching [40 points]

You and your colleague are tasked with designing the prefetcher of a machine your company is designing. The machine has a single core, L1 and L2 caches and a DRAM memory system.

We will examine different prefetcher designs and analyze the trade-offs involved.

- For all parts of this question, we want to compute prefetch accuracy, coverage and bandwidth overhead after the prefetcher is trained and is in steady state. Therefore, **exclude the first six requests from all computations**.
- If there is a request already outstanding to a cache block, a new request for the same cache block will not be generated. The new request will be merged with the already outstanding request in the MSHRs.

(a) You first design a stride prefetcher that observes the last three cache block requests. If there is a constant stride between the last three requests, it prefetches the next cache block using that stride.

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

```
A A+1 A+2 A+7 A+8 A+9 A+14 A+15 A+16 A+21 A+22 A+23 A+28 A+29 A+30...
```

Assume this pattern continues for a long time.

Compute the coverage of your stride prefetcher for this application.
0% After each group of three requests, a prefetch is triggered due to a detected stride, but the prefetched block is always useless; none of the demand requests are covered by this prefetch.

Compute the accuracy of your stride prefetcher for this application.
0%

(b) Your colleague designs a new prefetcher that, on a cache block access, prefetches the next N cache blocks.

The coverage and accuracy of this prefetcher are 66.67% and 50% respectively for the above application. What is the value of N?

$N = 2$.

After (for example) the access to block 14, the prefetcher prefetches blocks 15 and 16. After 15, it prefetches 16 (merged with the prefetch that was already issued) and 17. After 16, it prefetches 17 and 18. Hence, two out of every three demand accesses are covered (66.7%), and half of prefetches are useful (50%).

We define the bandwidth overhead of a prefetcher as

$$\frac{\text{Total number of cache block requests with the prefetcher}}{\text{Total number of cache block requests without the prefetcher}} \quad (1)$$

What is the bandwidth overhead of this next-N-block prefetcher for the above application?

5/3.

For every group of accesses to three consecutive cache blocks, two extra blocks are prefetched. For example, cache blocks 14, 15 and 16 are fetched. In addition to these, blocks 17 and 18 are prefetched.

(c) Your colleague wants to improve the coverage of her next-N-block prefetcher further for the above application, but is willing to tolerate a bandwidth overhead of at most 2x.
Is this possible? Why or why not?

To get better coverage, the prefetcher must prefetch into the next group of 3 strided requests from the previous group, because the full group of 3 is already prefetched by the first. For instance, on an access to A+14, A+15 and A+16 are already prefetched. To improve coverage, A+21 (which is the first of the next group of 3 strided requests) should be prefetched. However, this would require prefetching the four cache blocks in between A+16 and A+21 (A+17, A+18, A+19, A+20). This increases the bandwidth overhead beyond 2x.

(d) What is the minimum value of N required to achieve a coverage of 100% for the above application? Remember that you should exclude the first six requests from your computations.

$N = 5$ (so that A+16 prefetches A+21, then A+21 prefetches A+22, A+23, etc.)

What is the bandwidth overhead at this value of N?

7/3

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

1. A prefetcher that learns the pattern of strides: 1, 1, 5, in this case. This can be accomplished by keeping the last three strides and a confidence counter for each pattern of last three strides.

2. A *two-delta stride prefetcher* could record up to two different strides Δ_1 and Δ_2, and the number of strides Δ_1 that are traversed before a stride of Δ_2 is traversed. For this sequence, the prefetcher would learn that $\Delta_1 = 1$, $\Delta_2 = 5$, and that two strides of Δ_1 occur followed by one stride of Δ_2.

4 Memory System [40 points]

A machine with a 4 GB DRAM main memory system has 4 channels, 1 rank per channel and 4 banks per rank. The cache block size is 64 bytes.

(a) You are given the following byte addresses and the channel and bank to which they are mapped:

Byte: `0x0000` ⇒ Channel 0, Bank 0
Byte: `0x0100` ⇒ Channel 0, Bank 0
Byte: `0x0200` ⇒ Channel 0, Bank 0
Byte: `0x0400` ⇒ Channel 1, Bank 0
Byte: `0x0800` ⇒ Channel 2, Bank 0
Byte: `0x0C00` ⇒ Channel 3, Bank 0
Byte: `0x1000` ⇒ Channel 0, Bank 1
Byte: `0x2000` ⇒ Channel 0, Bank 2
Byte: `0x3000` ⇒ Channel 0, Bank 3

Determine which bits of the address are used for each of the following address components. Assume row bits are higher order than column bits:

- Byte on bus
  Addr [2 : 0]
- Channel bits (channel bits are contiguous)
  Addr [_____ : _____]
- Bank bits (bank bits are contiguous)
  Addr [_____ : _____]
- Column bits (column bits are contiguous)
  Addr [_____ : _____]
- Row bits (row bits are contiguous)
  Addr [_____ : _____]

(b) Two applications App 1 and App 2 share this memory system (using the address mapping scheme you determined in part (**a**)). The memory scheduling policy employed is FR-FCFS. The following requests are queued at the memory controller request buffer at time t. Assume the first request (A) is the oldest and the last one (A + 15) is the youngest.

A B A + 1 A + 2 A + 3 B + 10 A + 4 B + 12 A + 5 A + 6 A + 7
A + 8 A + 9 A + 10 A + 11 A + 12 A + 13 A + 14 A + 15

These are cache block addresses, not byte addresses. Note that requests to A + x are from App 1, while requests to B + x are from App 2. Addresses A and B are row-aligned (i.e., they are at the start of a row) and are at the same bank but are in different rows.

Assuming row-buffer hits take T time units to service and row-buffer conflicts/misses take 2T time units to service, what is the slowdown (compared to when run alone on the same system) of i) App 1? ii) App 2? 1

ii) App 2? (2T+15T+2T+2T)/(2T+2T) = 21/4

(c) Which application slows down more? Why? 2 Why?
High Row-buffer locality of application 1.

(d) In class, we discussed memory channel partitioning and memory request scheduling as two solutions to mitigate interference and application slowdowns in multicore systems. Propose another solution to reduce the slowdown of the more-slowed-down application, *without* increasing the slowdown of the other application? Be concrete.

5 Running Ahead [55 points]

Consider the following program, running on an in-order processor with no pipelining:

```
LD  R1  ← (R3)     // Load A
ADD R2  ← R4, R6
LD  R9  ← (R5)     // Load B
ADD R4  ← R7, R8
LD  R11 ← (R16)    // Load C
ADD R7  ← R8, R10
LD  R12 ← (R11)    // Load D
ADD R6  ← R8, R15
```

Assume that all registers are initialized and available prior to the beginning of the shown code. Each load takes 1 cycle to execute, and each add takes 2 cycles to execute. Loads A through D are all cache misses. In addition to the 1 cycle taken to execute each load, these cache misses take 6, 9, 12, and 3 cycles, respectively for Loads A through D, to complete. For now, assume that no penalty is incurred when entering or exiting runahead mode.

Note: Please show all your work for partial credit.

(a) For how many cycles does this program run *without* runahead execution?

42 cycles

Table 1: Execution timeline for Problem 6(b), using runahead execution with no exit penalty and no optimizations

Cycle	Operations		Enter/Exit Runahead Mode	Runahead Mode?	Is Additional Runahead Instruction?
1	Load 1 Issue		enter RA		
2	Add 1			Yes	Yes
3	Add 1			Yes	Yes
4	Load 2 Issue			Yes	Yes
5	Add 2			Yes	Yes
6	Add 2			Yes	Yes
7	Load 1 Finish	Load 3 Issue	exit RA	Yes	Yes
8	Add 1				
9	Add 1				
10	Load 2 Issue, already issued		enter RA		
11	Add 2			Yes	Yes
12	Add 2			Yes	Yes
13	Load 2 Finish	Load 3 Issue, already issued	exit RA	Yes	Yes
14	Add 2				
15	Add 2				
16	Load 3 Issue, already issued		enter RA		
17	Add 3			Yes	Yes
18	Add 3			Yes	Yes
19	Load 3 Finish	Load 4 Issue, dependent on R11 from Load 3. Load 3 finishes at the end of cycle 19, so R11 isn't available now.	exit RA	Yes	Yes
20	Add 3				
21	Add 3				
22	Load 4 Issue		enter RA		
23	Add 4			Yes	Yes
24	Add 4			Yes	Yes
25	Load 4 Finish		exit RA	Yes	
26	Add 4				
27	Add 4				

(b) For how many cycles does this program run *with* runahead execution?

27 cycles. See Table 1.

(c) How many additional instructions are executed in runahead execution mode?

14 instructions. See Table 1.

(d) Next, assume that exiting runahead execution mode incurs a penalty of 3 cycles. In this case, for how many cycles does this program run *with* runahead execution?

4 runahead periods in total. $27 + 4 * 3 = 39$ cycles.

(e) At least how many cycles should the runahead exit penalty be, such that enabling runahead execution decreases performance? Please show your work.

4 cycles. $(27 + 4 \times X > 42)$

(f) Which load instructions cause runahead periods? Circle all that did:

Load A Load B Load C Load D

For each load that caused a runahead period, tell us if the period generated a prefetch request. If it did not, explain why it did not and what type of a period it caused.

Load A: This period generated prefetches.

Load B: Short runahead period. / Overlapping runahead periods.

Load C: Dependent loads. / Short runahead period.

Load D:	No loads to prefetch in this period. (Useless runahead period)

(g) For each load that caused a runahead period that did not result in a prefetch, explain how you would best mitigate the inefficiency of runahead execution.

Load A:	N/A. Prefetch was generated.
Load B:	Do not enter runahead mode if only a few cycles are left before the corresponding cache miss will be filled. Do not enter runahead mode if the subsequent instruction(s) have already been executed during a previous runahead period.
Load C:	Predict the value of cache-miss address loads.
Load D:	Predict if a period will generate useful cache misses, and do not execute that period if it's deemed useless.

(h) If all useless runahead periods were eliminated, how many additional instructions would be executed in runahead mode?

Since only the first runahead period generated prefetches, only that runahead period will be executed. According to Table 1, 6 instructions are executed in the first runahead period.

How does this number compare with your answer from part (c)?

Less.

(i) Assume still that the runahead exit penalty is 3 cycles, as in part (d). If all useless runahead execution periods were eliminated (i.e., runahead execution is made *efficient*), for how many cycles does the program run with runahead execution?

One runahead period, so $27 + 1 * 3 = 30$ cycles.

How does this number compare with your answer from part (d)?

Less.

(j) At least how many cycles should the runahead exit penalty be such that enabling *efficient runahead execution* decreases performance? Please show your work.

16 cycles. $(27 + 1 \times X > 42)$