CMU 18-447 Introduction to Computer Architecture, Spring 2014
HW 7: Prefetching, Runahead, Cache Coherence, Multiprocessors, Faults

Instructor: Prof. Onur Mutlu
TAs: Rachata Ausavarungnirun, Varun Kohli, Xiao Bo Zhao, Paraj Tyle

Assigned: Wed., 4/9, 2014
Due: **Wed., 4/30, 2014 (Midnight)**
Handin: `/afs/ece/class/ece447/handin/hw7`
Please submit as ONE PDF: [andrewID].pdf

1 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.
Compute the accuracy of your stride prefetcher for this application.

(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?

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?

(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? **YES** **NO**
Why or why not?

(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.

What is the bandwidth overhead at this value of N?

(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 bandwidth overhead. Propose a prefetcher design that accomplishes this goal.
Be concrete and clear.

2 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?

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

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

(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?

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

(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.

(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.

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

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

(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?

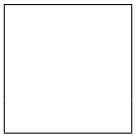
How does this number compare with your answer from part (d)?

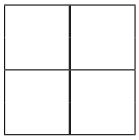
(j) At least how many cycles should the runahead exit penalty be such that enabling *efficient runahead execution* decreases performance? Please show your work.

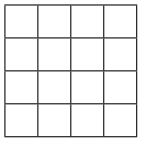
3 Amdahl's Law [30 points]

Consider the following three processors (X, Y, and Z) that are fabricated on a constant silicon area of $16A$. Assume that the single-thread performance of a core increases with the square root of its area.

Processor X	**Processor Y**	**Processor Z**
1 *large* core of area $16A$	4 *medium* cores of area $4A$	16 *small* cores of area A







On each of the three processors, we will execute a workload where S fraction of its work is serial, and where $1 - S$ fraction of its work is **infinitely** parallelizable. As a function of S, plot the execution time of the workload on each of the three processors. Assume that it takes time T to execute the entire workload using only one of the small cores of Processor Z.

Please label the value of the **y-axis intercept** and the **slope**.

(a) Which processor has the lowest execution time for the widest range of S?

(b) Typically, for a realistic workload, the parallel fraction is not infinitely parallelizable. What are the three fundamental reasons why?

4 Coherent Caches [40 points]

You just got back a prototype of your latest processor design, which has two cores and uses the MESI cache coherence protocol for each core's private L1 caches.

(a) For this subproblem, assume a 2-core processor where each core contains a private L1 cache. The two caches are coherent and implement the MESI protocol. Assume that each cache can only hold 1 cache block, and that both caches are holding the same cache block for the duration of this problem.

The letters below show the MESI state transitions of the cache block in CPU 0's private L1 cache.

For each transition (pair of adjacent MESI states) below, list *all possible* actions CPU 0 and CPU 1 could perform to cause that transition. The first transition, where the cache block in CPU 0's cache transitions from Invalid to Exclusive state, has been completed for you as an example.

I E S S I M M E S E I S M S

CPU 0 issued a read. CPU 1 issued nothing.

(b) **Scenario 1** Let's say that you discover that there are some bugs in the design of your processor's coherence modules. Specifically, the `BusRead` and `BusWrite` signals on the module occasionally do *not* get asserted when they should have (but data still gets transferred correctly to the cache).

Fill in the table below with a ✔ if, for each MESI state, the missing signal has no effect on correctness. If correctness may be affected, fill in a ✖.

	`BusRead`	`BusWrite`
M		
E		
S		
I		

(c) **Scenario 2** Let's say that instead you discover that there are no bugs in the design of your processor's coherence modules. Instead, however, you find that occasionally cosmic rays strike the MESI state storage in your coherence modules, causing a state to instantaneously change to another.

Fill in a cell in the table below with a ✔ if, for a starting MESI state on the top, instantaneously changing the state to the state on the left affects neither correctness nor performance. Fill in a cell in the table with a ◯ if correctness is not affected but performance could be affected by the state change. If correctness may be affected, fill in a ✖.

ending state (current status) \ starting state (real status)	m	e	s	i
m				
e				
s				
i				

5 Parallel Speedup [30 points]

You are a programmer at a large corporation, and you have been asked to parallelize an old program so that it runs faster on modern multicore processors.

(a) You parallelize the program and discover that its speedup over the single-threaded version of the same program is significantly less than the number of processors. You find that many cache invalidations are occuring in each core's data cache. What program behavior could be causing these invalidations (in 20 words or less)?

(b) You modify the program to fix this first performance issue. However, now you find that the program is slowed down by a global state update that must happen in only a single thread after every parallel computation. In particular, your program performs 90% of its work (measured as processor-seconds) in the parallel portion and 10% of its work in this serial portion. The parallel portion is perfectly parallelizable. What is the maximum speedup of the program if the multicore processor had an infinite number of cores?

(c) How many processors would be required to attain a speedup of 4?

(d) In order to execute your program with parallel and serial portions more efficiently, your corporation decides to design a custom heterogeneous processor.

- This processor will have one large core (which executes code more quickly but also takes greater die area on-chip) and multiple small cores (which execute code more slowly but also consume less area), all sharing one processor die.
- When your program is in its parallel portion, all of its threads execute **only** on small cores.
- When your program is in its serial portion, the one active thread executes on the large core.
- Performance (execution speed) of a core is proportional to the square root of its area.
- Assume that there are 16 units of die area available. A small core must take 1 unit of die area. The large core may take any number of units of die area n^2, where n is a positive integer.
- Assume that any area not used by the large core will be filled with small cores.

How large would you make the large core for the fastest possible execution of your program?

What would the same program's speedup be if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

Does it make sense to use a heterogeneous system for this program which has 10% of its work in serial sections?
YES **NO**

Why or why not?

(e) Now you optimize the serial portion of your program and it becomes only 4% of total work (the parallel portion is the remaining 96%). What is the best choice for the size of the large core in this case?

What is the program's speedup for this choice of large core size?

What would the same program's speedup be for this 4%/96% serial/parallel split if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

Does it make sense to use a heterogeneous system for this program which has 4% of its work in serial sections?
YES **NO**

Why or why not?

6 Transient Faults [45 points]

In class and in labs, we have implicitly assumed that the circuits in the processor are always *correct*. However, in the real world, this is not always the case. One common type of physical problem in a processor is called a *transient* fault. A transient fault simply means that a bit in a flip-flop or register incorrectly changes (i.e., flips) to the opposite state (e.g., from 0 to 1 or 1 to 0) temporarily.

Consider how a transient fault could affect each of the following processor structures. For each part of this question, answer

(i) does a transient fault affect the *correctness* of the processor (will the processor still give correct results for **any** program if such a fault occurs in this structure)?

(ii) *if the fault does not affect correctness*, does it affect the *performance* of the processor?

Valid answers for each question are "can affect" or "cannot affect". Answer for *performance* in a particular situation only if correctness is not affected. When in doubt, state your assumptions.

(a) A bit in the global history register of the global branch predictor:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

(b) A bit in a pipeline register (prior to the stage where branches are resolved) that when set to '1' indicates that a branch should trigger a flush due to misprediction, when that pipeline register is currently holding a branch instruction:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

(c) A bit in the state register of the microsequencer in a microcoded design:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

(d) A bit in the tag field of a register alias table (RAT) entry in an out-of-order design:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

(e) The dirty bit of a cache block in a write-back cache:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

(f) A bit in the application id field of an application-aware memory scheduler's request buffer entry:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why?

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why? [ ]

Performance: CAN AFFECT CANNOT AFFECT

(g) Reference bit in a page table entry:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why? [ ]

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why? [ ]

Performance: CAN AFFECT CANNOT AFFECT

(h) A bit indicating the processor is in runahead mode:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why? [ ]

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why? [ ]

Performance: CAN AFFECT CANNOT AFFECT

(i) A bit in the stride field of a stride prefetcher:

Flipped from 0 to 1:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why? [ ]

Performance: CAN AFFECT CANNOT AFFECT

Flipped from 1 to 0:

Correctness: CAN AFFECT CANNOT AFFECT

If correctness is affected, why? [ ]

Performance: CAN AFFECT CANNOT AFFECT

7 Cache coherence: MESI [20 points]

Assume you are designing a MESI snoopy bus cache coherence protocol for write-back private caches in a multi-core processor. Each private cache is connected to its own processor core as well as a common bus that is shared among other private caches.

There are 4 input commands a private cache may get for a cacheline. Assume that bus commands and core commands will not arrive at the same time:

- CoreRd: Read request from the cache's core
- CoreWr: Write request from the cache's core
- BusGet: Read request from the bus
- BusGetI: Read and Invalidate request from the bus (invalidates shared data in the cache that receives the request)

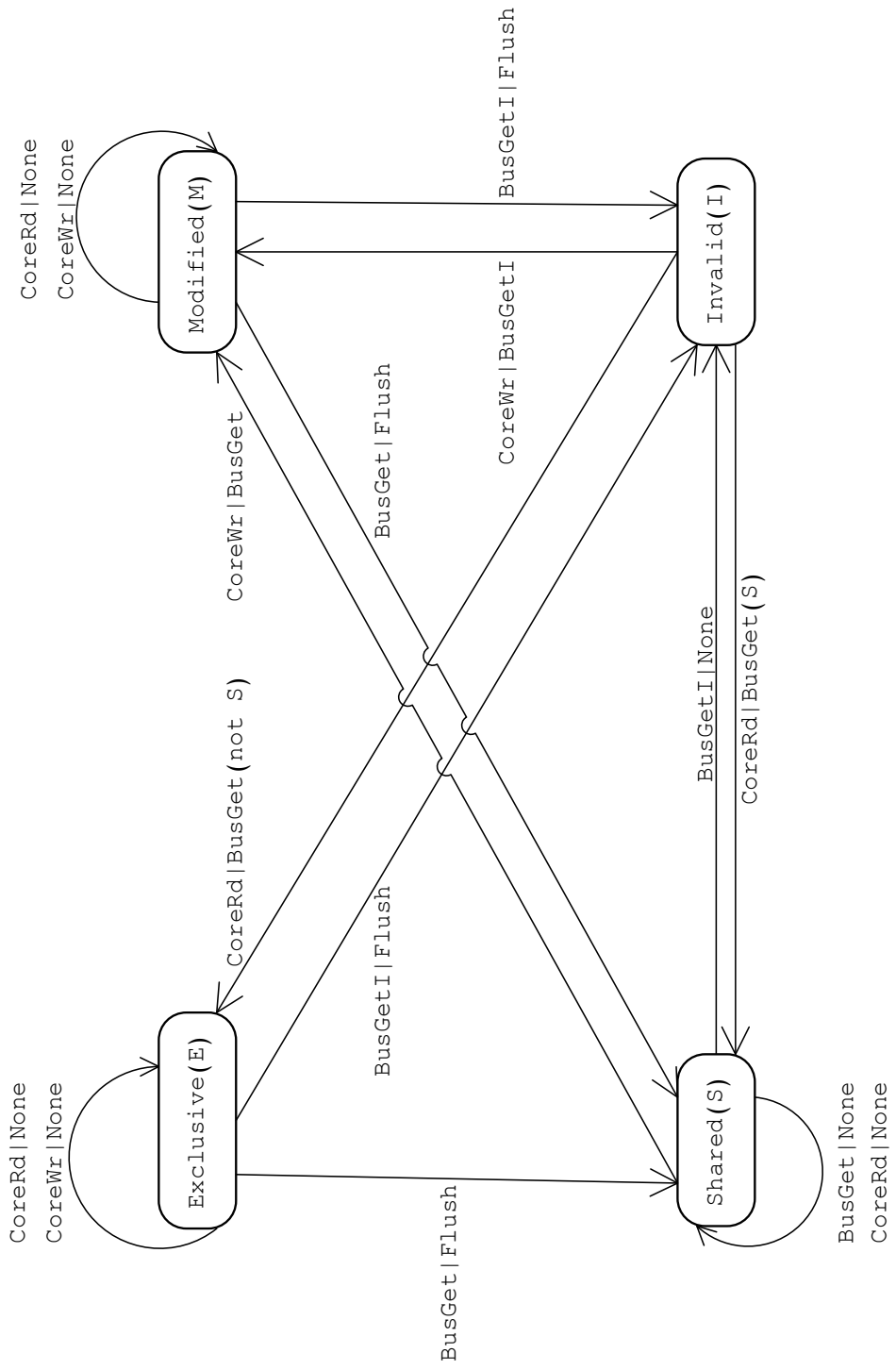
There are 4 actions a cache can take while transferring to another state:

- Flush: Write back the cacheline to lower-level cache
- BusGet: Issue a BusGet request to the bus
- BusGetI: Issue a BusGetI request to the bus
- None: Take no action

There is also an "is_shared (S)" signal on the bus. S is asserted upon a BusGet request when at least one of the private caches shares a copy of the data (BusGet (S)). Otherwise S is deasserted (BusGet (not S)).

Assume upon a BusGet or BusGetI request, the inclusive lower-level cache will eventually supply the data and there are no private cache to private cache transfers.

On the next page, Rachata drew a MESI state diagram. There are 4 mistakes in his diagram. **Please show the mistakes and correct them.** You may want to practice on the scratch paper first before finalizing your answer. If you made a mess, clearly write down the mistakes and the changes below.


Modified(M)
Invalid(I)
Exclusive(E)
Shared(S)
CoreRd|None
CoreWr|None
CoreRd|None
CoreWr|None
BusGet|None
CoreRd|None
BusGetI|Flush
CoreWr|BusGet
BusGet|Flush
CoreWr|BusGetI
CoreRd|BusGet(not S)
BusGetI|Flush
BusGetI|None
CoreRd|BusGet(S)
BusGet|Flush