ETH 263-2210-00L Computer Architecture, Fall 2022
HW 4: Memory QoS, Prefetching, Memory Consistency, Cache Coherence (SOLUTIONS)

Instructor: Prof. Onur Mutlu
TAs: Juan Gómez Luna, Mohammad Sadrosadati, Mohammed Alser, Rahul Bera, Nisa Bostanci, João Dinis Ferreira, Can Firtina, Nika Mansouri Ghiasi, Geraldo Francisco De Oliveira Junior, Konstantinos Kanellopoulos, Joël Lindegger, Rakesh Nadig, Ataberk Olgun, Abdullah Giray Yaglikci, Yahya Can Tugrul, Haocong Luo, Banu Cavlak, Aditya Manglik

Given: Monday, November 28, 2022
Due: **Friday, December 9, 2022**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture22/`. Please, check your inbox, you should have received an email with the password you should use to login. If you did not receive any email, contact comparch@lists.inf.ethz.ch. In the first page after login, you should click in "Computer Architecture Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-6).** You should upload your answers to the Moodle Platform (`https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=831337`) as a single PDF file.
- If you have any questions regarding this homework, please ask them the Moodle forum (`https://moodle-app2.let.ethz.ch/mod/moodleoverflow/view.php?id=831341`).
- Please note that the handin questions have a hard deadline. However, you can submit your paper reviews till January 31 2023.

1. Critical Paper Reviews [1,000 points]

We assign you five **required readings** for this homework. You may access them by simply clicking on the QR codes below or scanning them.

Required 1 Required 2 Required 3 Required 4 Required 5

Write an approximately one-page critical review for the readings (i.e., papers from #1 to #5). If you review a paper other than the 5 mandatory papers, you will receive 200 BONUS points on top of 1,000 points you may get from paper reviews (i.e., each additional submission is worth 200 BONUS points with a possibility to get up to 6400 points). Note that you will get **zero** points from the critical paper reviews if you do not submit the required paper reviews (i.e., papers from #1 to #5).

Please read the guideline slides for reviewing papers and watch Prof. Mutlu's guideline video on how to do a critical paper review. We also provide you with sample reviews which you can access using the QR code. A review with bullet point style is more appreciated. Try not to use very long sentences and paragraphs. Keep your writing and sentences simple. Make your points bullet by bullet, as much as possible. **We will give out extra credit that is worth 0.5% of your total course grade for each good review.**

Guideline Slides Guideline Video Sample Reviews

1. **(REQUIRED)** Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors", HPCA 2003, `https://people.inf.ethz.ch/omutlu/pub/mutlu_hpca03.pdf`
2. **(REQUIRED)** Hashemi et al., "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads", MICRO 2016, `https://people.inf.ethz.ch/omutlu/pub/continuous-runahead-engine_micro16.pdf`
3. **(REQUIRED)** Subramanian et al., "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling", TPDS 2016, `https://people.inf.ethz.ch/omutlu/pub/bliss-memory-scheduler_ieee-tpds16.pdf`
4. **(REQUIRED)** Ebrahimi et al., "Fairness via Source Throttling: A Configurable and High- Performance Fairness Substrate for Multi-Core Memory Systems", ASPLOS 2010, `https://people.inf.ethz.ch/omutlu/pub/fst_asplos10.pdf`
5. **(REQUIRED)** Boroumand et al., "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators", ISCA 2019, `https://people.inf.ethz.ch/omutlu/pub/CONDA-coherence-for-near-data-accelerators_isca19.pdf`
6. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Transactions on Computers 1979, `https://safari.ethz.ch/architecture/fall2021/lib/exe/fetch.php?media=multiprocessors-multicomputers.pdf`
7. Papamarcos et al., "A low-overhead coherence solution for multiprocessors with private cache memories", ISCA 1984, `https://safari.ethz.ch/architecture/fall2022/lib/exe/fetch.php?media=papamarcos-isca84.pdf`
8. Dubois et al., "Memory Access Buffering in Multiprocessors", ISCA 1986, `https://safari.ethz.ch/architecture/fall2022/lib/exe/fetch.php?media=dubois-isca1986.pdf`
9. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", ISCA 1990, `https://safari.ethz.ch/architecture/fall2022/lib/exe/fetch.php?media=gharachorloo-isca90.pdf`
10. Gharachorloo et al., "Two Techniques to Enhance the Performance of Memory Consistency Models", ICPP 1991, `https://safari.ethz.ch/architecture/fall2022/lib/exe/fetch.php?media=gharachorloo-icpp91.pdf`
11. Laudon et al., "The SGI Origin: a ccNUMA Highly Scalable Server", ISCA 1997, `https://safari.ethz.ch/architecture/fall2022/lib/exe/fetch.php?media=laudon-isca97.pdf`
12. Mutlu et al., "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns", MICRO 2005, `https://people.inf.ethz.ch/omutlu/pub/mutlu_micro05.pdf`
13. Mutlu et al., "Techniques for Efficient Processing in Runahead Execution Engines", ISCA 2005, `https://people.inf.ethz.ch/omutlu/pub/mutlu_isca05.pdf`
14. Moscibroda et al., "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems", USENIX Security 2007, `https://people.inf.ethz.ch/omutlu/pub/mph_usenix_security07.pdf`
15. Ceze et al., "BulkSC: Bulk Enforcement of Sequential Consistency", ISCA 2007, `https://safari.ethz.ch/architecture/fall2022/lib/exe/fetch.php?media=isca07_bulksc.pdf`
16. Mutlu et al., "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors", MICRO 2007, `https://people.inf.ethz.ch/omutlu/pub/stfm_micro07.pdf`
17. Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers", HPCA 2007, `https://people.inf.ethz.ch/omutlu/pub/srinath_hpca07.pdf`
18. Mutlu et al., "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems", ISCA 2008, `https://people.inf.ethz.ch/omutlu/pub/parbs_isca08.pdf`
19. Lee et al., "Prefetch-Aware DRAM Controllers", MICRO 2008, `https://people.inf.ethz.ch/omutlu/pub/prefetch-dram_micro08.pdf`
20. Ebrahimi et al., "Coordinated Control of Multiple Prefetchers in Multi-Core Systems", MICRO 2009, `https://people.inf.ethz.ch/omutlu/pub/coordinated-prefetching_micro09.pdf`
21. Ebrahimi et al., "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems", HPCA 2009, `https://people.inf.ethz.ch/omutlu/pub/bandwidth_lds_hpca09.pdf`
22. Cain et al., "Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor", ISPASS 2010, `https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=cain-ispass-2010.pdf`
23. Kim et al., "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers", HPCA 2010, `https://people.inf.ethz.ch/omutlu/pub/atlas_hpca10.pdf`
24. Kim et al., "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior", MICRO 2010, `https://people.inf.ethz.ch/omutlu/pub/tcm_micro10.pdf`
25. Ebrahimi et al., "Parallel Application Memory Scheduling", MICRO 2011, `https://people.inf.ethz.ch/omutlu/pub/parallel-memory-scheduling_micro11.pdf`

26. Ebrahimi et al., "Prefetch-Aware Shared Resource Management for Multi-Core Systems", ISCA 2011, https://people.inf.ethz.ch/omutlu/pub/prefetchaware-shared-resources_isca11.pdf
27. Ausavarungnirun et al., "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems", ISCA 2012, https://people.inf.ethz.ch/omutlu/pub/staged-memory-scheduling_isca12.pdf
28. Jog et al., "Orchestrated Scheduling and Prefetching for GPGPUs", ISCA 2013, https://people.inf.ethz.ch/omutlu/pub/orchestrated-gpgpu-scheduling-prefetching_isca13.pdf
29. Seshadri et al., "Mitigating Prefetcher-Caused Pollution using Informed Caching Policies for Prefetched Blocks", TACO 2015, https://people.inf.ethz.ch/omutlu/pub/informed-caching-for-prefetching_taco15.pdf
30. Lee et al., "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM", PACT 2015, https://people.inf.ethz.ch/omutlu/pub/decoupled-dma_pact15.pdf
31. Bera et al., "Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning", MICRO 2021, https://people.inf.ethz.ch/omutlu/pub/Pythia-customizable-hardware-prefetcher-using-reinforcement-learning_micro21.pdf
32. Bera et al., "Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction", MICRO 2022, https://arxiv.org/pdf/2209.00188v3.pdf

2. QoS-aware Memory Scheduling [200 points]

In this problem, we will look at the following three different memory-scheduling policies that we studied in class:

1. FR-FCFS: First-Ready, First-Come-First-Served,
2. STFM: Stall-Time Fair Memory scheduling,
3. PAR-BS: PARallelism-aware Batch Scheduling.

(a) The FR-FCFS policy is a commonly-used memory scheduling policy that aims to maximize DRAM throughput. Explain how the FR-FCFS policy can improve DRAM throughput compared to the FCFS (first-come-first-served) policy **with an example DRAM row access pattern(s)**.

FR-FCFS improves DRAM throughput by maximizing the row-hit ratio (i.e., minimizing the row-miss penalty). In the following memory-access pattern, FR-FCFS first serves memory requests to `Row 0` without re-activating the row. On the other hand, FCFS repeats activating the two rows alternately, which significantly increases the latency of each request.

Row access pattern: `Row 0 - Row 1 - Row 0 - Row 1 - ...`

(b) While the FR-FCFS policy provides higher DRAM throughput over the FCFS policy, it makes the DRAM memory system vulnerable to denial of service (DoS) attacks that significantly degrade a thread's performance when multiple threads share the DRAM memory system. Consider two memory-intensive threads A and B that exhibit the following memory-access patterns:

- A exhibits a streaming access pattern that sequentially accesses data in the same rows.
- B exhibits a random access pattern that randomly accesses data in different rows.
- A and B are similarly memory intensive, i.e., they generate approximately the same number of memory requests within a given time frame.

Under the FR-FCFS policy, which thread of A or B can be used by an adversary to perform DoS attacks to the other thread? Justify your answer.

A

Explanation: When A and B share a DRAM memory system that adopts the FR-FCFS policy, the memory controller prefers to serve memory requests from A, as they likely hit the row buffer.

(c) The STFM policy aims to achieve fair memory scheduling that equalizes the slowdown of equal-priority threads relative to when each thread is run alone on the same system. How does the STFM policy compute the slowdown of a thread?

For each thread, an STFM-enabled memory controller tracks ST_{shared}, the DRAM-related stall-time when the thread runs with other threads, and estimates ST_{alone}, the DRAM-related stall-time when the thread runs alone. At each cycle, the DRAM controller computes a thread's slowdown, defined as follows:

$$Slowdown = ST_{\text{shared}}/ST_{\text{alone}}.$$

(d) How does the STFM policy equalize the slowdown of threads using the computed slowdown values?

Let S_i be the slowdown of the i-th thread among N threads that run together in the same system. At each cycle, the memory controller computes the current unfairness of the DRAM memory system which is defined as follows:

$$Unfairness = max(\{S_i|1 \leq i \leq N\})/min(\{S_i|1 \leq i \leq N\}).$$

If unfairness is less than a threshold, the memory controller uses a DRAM-throughput-oriented scheduling policy. Otherwise, it uses a fairness-oriented policy that prioritizes requests from the thread with the largest slowdown, as defined above.

(e) The PAR-BS policy aims to enhance both the performance and fairness of shared DRAM systems. The first principle of the PAR-BS policy is to schedule requests from a thread (to different banks) back to back, which preserves each thread's bank-level parallelism. What is the potential problem (described in class) when the memory controller only adopts the first principle? To get the full credit, your answer should explain when the problem can occur.

It can lead to starvation for threads that only issue memory requests to a single bank, while other threads issue memory requests to different banks.

(f) Explain request batching, the second principle of the PAR-BS policy that addresses the potential problem identified in the previous question. To get the full credit, be specific in your explanation (e.g., an answer like "The PAR-BS policy performs memory request in a batched manner" will get no points).

- The PAR-BS policy groups a fixed number of the oldest requests from each thread into a "batch".
- It services the batch before all other requests.
- It forms a new batch only when the current one is done.

3. Prefetching using Reinforcement Learning [200 points]

You are designing a hardware prefetcher for a processor using a reinforcement learning (RL) agent, as discussed in lecture about the Pythia prefetcher. For a memory request to cacheline address A, the prefetcher selects a prefetch offset O and issues a prefetch memory request to cacheline address $A+O$. For every prefetch request, the memory hierarchy provides a numerical reward R to the prefetcher that can take a value of one of the following three reward levels:

- Accurate (R_A), signifying that the prefetch request was demanded by the processor.
- Inaccurate (R_{IN}), signifying that the prefetch request was not demanded by the processor.
- No-prefetch (R_{NP}), signifying that the prefetcher did not prefetch anything.

In the initial configuration of the prefetcher, you set the values of the reward levels as follows: $R_A = 20$, $R_{IN} = -4$, and $R_{NP} = -10$.

Recall that the **coverage** of a prefetcher is defined as the fraction of a program's memory requests correctly prefetched by the prefetcher, while the **accuracy** of a prefetcher is defined as the fraction of prefetched requests that are actually demanded by the program.

(a) Which of the following statements, if any, are **CORRECT** if you set $R_{NP} = 1000$ in the initial prefetcher configuration? All other reward level values, except R_{NP}, remain the same as the initial configuration. Select **ALL** that apply and explain briefly. You may get partial credits for a partially-complete answer, given a correct explanation.

A. The coverage of the prefetcher will significantly increase.
B. The coverage of the prefetcher will significantly decrease.
C. The prefetcher will start prefetching aggressively.
D. The accuracy of the prefetcher may increase.

(B) and (D).

Setting $R_{NP} \gg R_A$ will strongly encourage the prefetcher not to prefetch. As a result, the prefetcher's coverage will drop significantly. However, the accuracy of the prefetcher may increase, as the number of generated prefetch requests will significantly reduce.

(b) Which of the following statements, if any, are **CORRECT** if you set $R_{IN} = -500$ in the initial prefetcher configuration? All other reward level values, except R_{IN}, remain the same as the initial configuration. Select **ALL** that apply and explain briefly. You may get partial credits for a partially-complete answer, given a correct explanation.

A. The prefetcher will aggressively prefetch, even though the prefetches might be inaccurate.
B. The accuracy of the prefetcher will likely increase.
C. The accuracy of the prefetcher will likely decrease.
D. The coverage of the prefetcher might increase or decrease.

(B) and (D).

Setting $R_{IN} \ll R_{NP}$ will strongly discourage the prefetcher to generate any prefetch request that might not be accurate. As a result, the accuracy of the prefetcher will likely increase. However, the coverage of the prefetcher might increase or decrease.

(c) You want to make the prefetcher system-aware by incorporating into the decision making process the type of power source used in the system. You want to configure the prefetcher in the following way:

i. The prefetcher should generate accurate prefetches whenever possible, irrespective of whether the system is connected to an external power adapter or running on a battery.

ii. If the system is connected to an external power adapter, the prefetcher should continue to prefetch, even if the prefetch might be inaccurate.

iii. If the system is running on battery, the prefetcher should prefer not to generate a prefetch request if the prefetch is likely to be inaccurate.

How would you configure the values of the three reward levels? The relative ordering of the reward level values is sufficient for a correct answer, rather than their exact values. Fill in the blanks below and explain your reasoning briefly in the provided box.

```
if(power_source == battery)
```

$\boxed{R_{IN}} < \boxed{R_{NP}} < \boxed{R_{A}}$

```
else if (power_source == external_adapter)
```

$\boxed{R_{NP}} < \boxed{R_{IN}} < \boxed{R_{A}}$

When the system is running on a battery, we set $R_{IN} < R_{NP} < R_A$, because we want the prefetcher to prefer not to prefetch rather than inaccurate prefetching. When the system is connected to an external power adapter, we set $R_{NP} < R_{IN} < R_A$, because we want the prefetcher to continue prefetching, even if the prefetch might be inaccurate.

4. Runahead Execution [200 points]

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

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

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

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

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

$61 < X < 93$.

Explanation. The program makes 100 memory accesses in total. To have 66 cache hits, a cache miss needs to be followed by 2 cache hits. Hence, the Runahead engine needs to prefetch 2 cache blocks. After getting a cache miss and entering Runahead mode, the processor needs to execute 30 instructions to reach the next LD instruction. To reach the LD instruction 2 times, the processor needs to execute at least 62 instructions (2*(29 ALU + 1 Branch + 1 LD)) in Runahead mode. If the processor spends more than 92 cycles in Runahead mode, then it will prefetch 3 cache lines instead of two, which will cause the number of cache hits to be different. Thus, the answer is $61 < X < 93$.

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

Yes, for $X < 31$ and $X > 123$.

Explanation. When X is equal to or smaller than 30 cycles, the processor will be in Runahead mode for insufficient amount of time to reach the next LD instruction (i.e., the next cache miss). Thus, none of the data will be prefetched and all memory accesses will get cache miss.
When X is larger than 123 cycles, the processor will prefetch 4 cache blocks. Since the prefetched cache blocks will map to the same cache set, the latest prefetched cache block will evict the first prefetched cache block in Runahead mode (note that the cache block that triggered Runahead execution remains in the cache due to the cache block replacement policy). This will cause a cache miss in the next iteration after leaving the Runahead mode. Thus, the accesses in the program will always miss in the cache.

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

25 cache misses.

Explanation. When $92 < X < 124$, the Runahead engine will prefetch exactly 3 cache blocks that will be accessed after leaving the Runahead mode. It is the minimum number of misses that could be achieved since all cache blocks accessed by the program map to the same cache set and the cache is 4-way associative.

5. Cache Coherence [200 points]

We have a system with 4 processors {P0, P1, P2, P3} that can access memory at byte granularity. Each processor has a private data cache with the following characteristics:

- Capacity of 256 bytes
- Direct-mapped
- Write-back
- Block size of 64 bytes

Each processor has also a dedicated private cache for instructions. The characteristics of the instruction caches are not necessary to solve this question.

All data 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. The range of accessible memory addresses is from `0x00000` *to* `0xfffff`.

The semantics of the instructions used in this question is the following:

Opcode	Operands	Description
ld	rx,[ry]	rx ← Mem[ry]
st	rx,[ry]	rx → Mem[ry]
addi	rx,#VAL	rx ← rx + VAL
j	TARGET	jump to TARGET
beq	rx,ry,TARGET	if([rx]==[ry]) jump to TARGET

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

P0
```
0        ld r1,[r2]
1        addi r1,#1
2        st r1,[r2]
3  LP :  ld r1,[r2]
4        beq r1,r4,END
5        j LP
6  END:  st r4, [r2]
```

P1
```
0  LP :  ld r1,[r4]
1        beq r1,r3,END
2        j LP
3  END:  addi r1,#1
4        st r1,[r4]
-
-
```

P2
```
0  LP :  ld r1,[r5]
1        beq r1,r3,END
2        j LP
3  END:  addi r1,#1
4        st r1,[r5]
-
-
```

P3
```
0  LP :  ld r1,[r2]
1        beq r1,r3,END
2        j LP
3  END:  addi r1,#1
4        st r1,[r2]
-
-
```

The initial state of the caches is unknown. After an arbitrarily large amount of time, all cores finish executing their code. The final tag store state of each data cache is as follows:

Final Tag Store States

Cache for P0

Set	*Tag*	*MESI state*
0	0x100	M
1	0xfff	M
2	0x010	S
3	0x110	I

Cache for P1

Set	*Tag*	*MESI state*
0	0x100	I
1	0xfff	I
2	0x010	S
3	0x110	I

Cache for P2

Set	*Tag*	*MESI state*
0	0x100	I
1	0xfff	I
2	0x011	E
3	0x110	S

Cache for P3

Set	*Tag*	*MESI state*
0	0x100	I
1	0xff1	S
2	0x010	I
3	0x10f	S

(a) What are the initial values of the registers in each of the 4 processors that ensure that the above final tag store states are deterministic (i.e., the final states are independent of the order in which the memory requests are issued to memory)? Explain your answer.

Solution:
P0: r1: X1, r2: 0x100YZ, r4: C+4
P1: r1: X2, r4: 0x100YZ, r3: C+1
P2: r1: X3, r5: 0x100YZ, r3: C+2
P3: r1: X4, r2: 0x100YZ, r3: C+3

Where X1, X2, X3, and X4 can be any value (these values are overwriten), Y is an integer number between 0x0 and 0x3, Z is an integer number between 0x0 and 0xF, and C is the initial content of the memory address 0x100YZ. The values of r3 in P1, P2, and P3 are interchangeable (e.g., the next r3 values are also valid: P1: C+2, P3: C+1, P2: C+3).

Explanation:
We need to find a solution that 1) does not have infinite loops (i.e., the program finish execution), 2) the final state of the tag store is deterministic, and 3) the final states are the ones provided in figure.We observe that 1) each individual thread reads and writes a unique memory position, and 2) all threads write to memory at the end of the execution. Because only P0 has cache blocks in Modified estate, we can conclude that all the other threads access to the same cache block. Otherwise, there would be some modified block in P1, P2, or P3. Because the only cache block that is present in all processors is the block in set 0, we conclude that the content of r2 in P0, r4 in P1, r5 in P2, and r2 in P3 are the same, and it is equal to a memory address that maps to Set 0, tag 0x100. Because Set 0, Tag 0x100 is only valid and with M state in P0, we conclude that P0 should be always the last processor to execute the last store instruction. We can ensure this scenario by forcing the termination of the programs of all processors in sequence. To this end, all processors need to access exactly the same byte in the cache block, i.e., all processors should access the same address 0x100YZ, where Y is an integer number between 0x0 and 0x3 (the 2 most significant bits of Y represent the set 0), and Z is an integer number between 0x0 and 0xF.
The content of r3 in P1, P2 and P3 should contain consecutive values. For example, these are some valid values for r3 in different processors: P1: C+1, P2: C+2, P3: C+3, and these are some valid values: P3: C+1, P1: C+2, P2: C+3. In this way, we ensure that all threads finish in the same deterministic order. Finally, r4 in P0 should have the value C+4 to ensure that it is the last processor that writes to memory.

(b) Fill in the following tables with the initial tag store states (i.e., Tag and MESI state) before having executed the 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

Set	*Tag*	*MESI state*
0	X	M, E, S, I
1	0xfff	M
2	0x010	S
3	0x110	I

Cache for P1

Set	*Tag*	*MESI state*
0	X	M, E, S, I
1	0xfff	I
2	0x010	S
3	0x110	I

Cache for P2

Set	*Tag*	*MESI state*
0	X	M, E, S, I
1	0xfff	I
2	0x011	E
3	0x110	S

Cache for P3

Set	*Tag*	*MESI state*
0	X	M, E, S, I
1	0xff1	S
2	0x010	I
3	0x10f	S

6. Memory Consistency [200 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] = 2;`	Instr. T1.0	`X[0] = 1;`
Instr. T0.1	`flag[0] = 1;`	Instr. T1.1	`X[0] += 2;`
Instr. T0.2	`a = X[0]*2;`	Instr. T1.2	`while(flag[0] == 1);`
Instr. T0.3	`b = Y[0]-1;`	Instr. T1.3	`a = flag[0];`
Instr. T0.4	`c = X[0];`	Instr. T1.4	`X[0] = 2;`
		Instr. T1.5	`Y[0] = 10;`

`X` and `flag` have been allocated in main memory. Thread 0 and Thread 1 have their private processor registers to store the values of `a` , `b`, and `c`. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 1. Assume each line of the C code segment of a thread is a single instruction.

(a) Do you find something that could be wrong in the C code segments? Explain your answer.

Thread 1 will never finish.

Explanation:
The while loop in instruction T1.2 is an infinite loop, because the value of `flag[0]` is 1 since the beginning of the program.

(b) What could be possible final values of `X[0]` in the SC processor, after executing both C code segments? Explain your answer. Provide all possible values.

2, 3, or 4.

Explanation:
The sequential consistency model ensures that the operations of each individual thread are executed in the order specified by its program. Across threads, the ordering is enforced by the use of `flag[0]`. Thread 1 will remain in instruction T1.2 until `flag[0]` has a value that is not 1. However, thread 1 will never finish execution. There are at least three possible sequentially-consistent orderings that lead to at most three different values of `X` at the end:
Ordering 1: T1.0 $\rightarrow$ T1.1 $\rightarrow$ T0.0, Final value: `X[0]` $= 2$.
Ordering 2: T0.0 $\rightarrow$ T1.0 $\rightarrow$ T1.1, Final value: `X[0]` $= 3$.
Ordering 3: T1.0 $\rightarrow$ T0.0 $\rightarrow$ T1.1, Final value: `X[0]` $= 4$.

(c) What could be possible final values of `a` in the SC processor, after executing both C code segments? Explain your answer. Provide all possible values.

2, 4, 6, or 8.

Explanation:
The value of `a` is twice the value of `X[0]`:
Ordering 1: T1.0 → T1.1 → T0.0 → T0.2, Final value: `X[0]` = 4.
Ordering 2: T0.0 → T1.0 → T1.1 → T0.2, Final value: `X[0]` = 6.
Ordering 3: T1.0 → T0.0 → T1.1 → T0.2, Final value: `X[0]` = 8.

Ordering 4: T0.0 → T0.1 → T1.0 → T0.2, Final value: `X[0]` = 2.

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

0.

Explanation:
Because the value of `b` depends only on the value of `Y[0]` (instruction T0.3). The initial value of `Y[0]` is 1. Instruction T1.4 will not be executed as T1 enters an infinite loop after executing instruction T1.2.

(e) With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called NC, that does not implement memory consistency. Thus, there is no guarantee on the ordering of instructions as seen by different cores.
What is the final value of `X[0]` in the NC processor, after executing both threads? Explain your answer.

1, 2, 3, or 4.

Explanation:
Since there is no guarantee of a strict order of memory operations, as seen by different cores, instruction T1.1 could complete before or after instruction T1.0, from the perspective of the core that executes thread 0. If instruction T1.1 completes before instruction T1.0, from the perspective of the core that executes T0, instruction T0.0 could complete before or after instruction T1.0. Thus, there are at least five possible weakly-consistent orderings that lead to different values of `X[0]` at the end:
Ordering 1: T0.0 $\rightarrow$ T1.1 $\rightarrow$ T1.0, Final value: `X[0]` = 1.
Ordering 2: T1.4 $\rightarrow$ T0.0 $\rightarrow$ T1.1 $\rightarrow$ T1.0, Final value: `X[0]` = 1.
Ordering 3: T1.0 $\rightarrow$ T1.1 $\rightarrow$ T0.1, Final value: `X[0]` = 2.
Ordering 4: T0.0 $\rightarrow$ T1.0 $\rightarrow$ T1.1, Final value: `X[0]` = 3.
Ordering 5: T1.0 $\rightarrow$ T0.0 $\rightarrow$ T1.1, Final value: `X[0]` = 4.