Computer Architecture

Lecture 19b: Memory Ordering (Memory Consistency)

Prof. Onur Mutlu
ETH Zürich

Fall 2021

2 December 2021

Recall: Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - "Embarrassingly parallel" applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs

Performance vs. Correctness

- Two metrics that are fundamentally at odds with each other
- You can always improve performance at the expense of correctness
  - Forget some critical lock in your program…
  - Design your architecture to ignore ordering of operations…
- We will see examples of this in fundamental support for multiprocessor operation (MIMD machines)
  - Memory ordering (consistency)
  - Cache coherence
- There is sometimes a real tradeoff between perf & correctness
  - When the application/user can tolerate the resulting "errors"
  - Recall EDEN (MICRO'19), Heterogeneous Reliability Memory (DSN'14)

EDEN: Exploiting Perf-Correctness Tradeoff

- Skanda Koppula, Lois Orosa, A. Giray Yaglikci, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu,
  "EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM"
  *Proceedings of the 52nd International Symposium on Microarchitecture (**MICRO**)*, Columbus, OH, USA, October 2019.
  [Slides (pptx) (pdf)]
  [Lightning Talk Slides (pptx) (pdf)]
  [Poster (pptx) (pdf)]
  [Lightning Talk Video (90 seconds)]
  [Full Talk Lecture (38 minutes)]

EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM

Skanda Koppula Lois Orosa A. Giray Yağlıkçı
Roknoddin Azizi Taha Shahroodi Konstantinos Kanellopoulos Onur Mutlu

ETH Zürich

More on Heterogeneous-Reliability Memory

- Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu, **"Characterizing Application Memory Error Vulnerability to Optimize Data Center Cost via Heterogeneous-Reliability Memory"** *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (**DSN**)*, Atlanta, GA, June 2014. [Summary] [Slides (pptx) (pdf)] [Coverage on ZDNet]

Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory

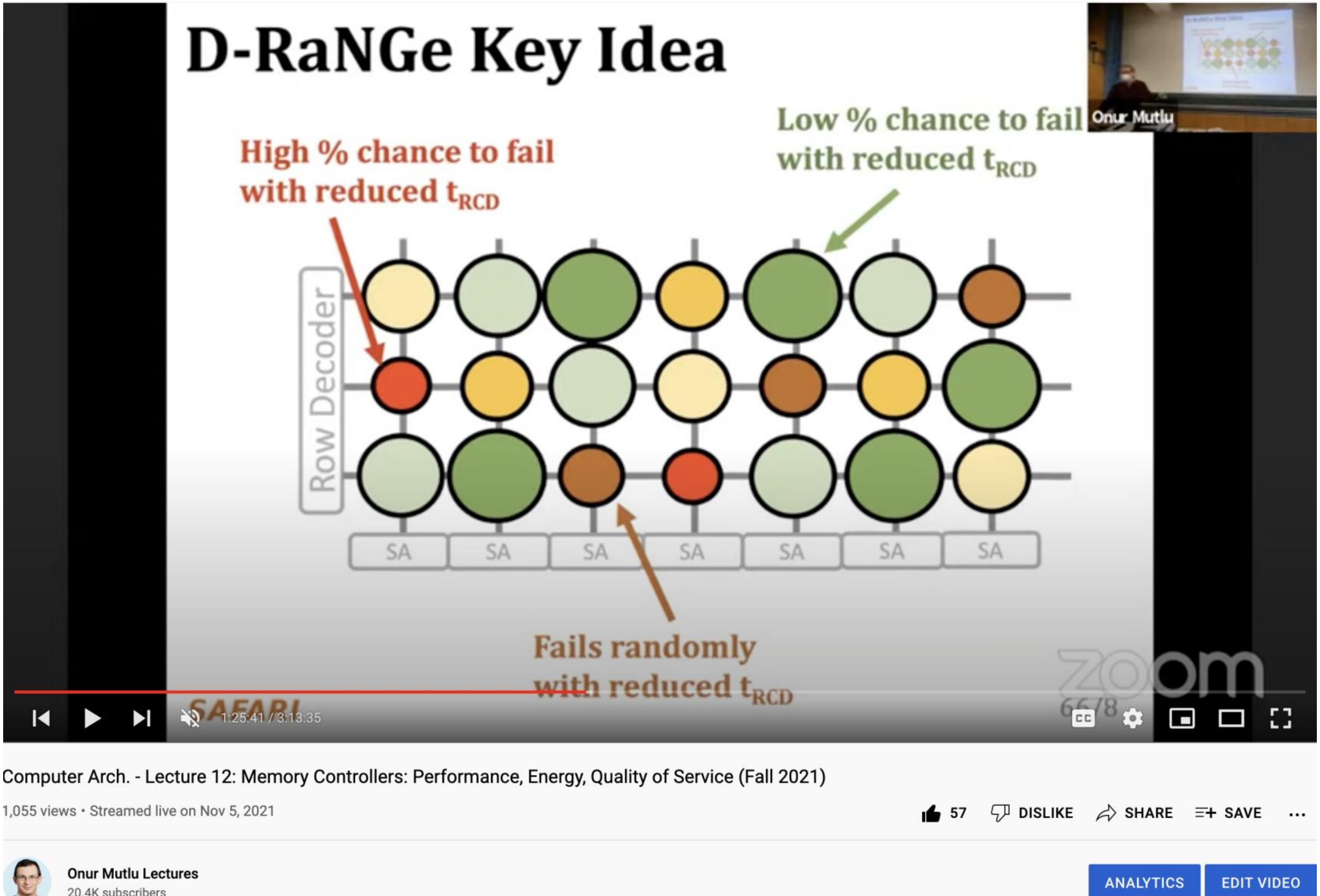
Yixin Luo Sriram Govindan* Bikash Sharma* Mark Santaniello* Justin Meza
Aman Kansal* Jie Liu* Badriddine Khessib* Kushagra Vaid* Onur Mutlu
Carnegie Mellon University, yixinluo@cs.cmu.edu, {meza, onur}@cmu.edu
*Microsoft Corporation, {srgovin, bsharma, marksan, kansal, jie.liu, bkhessib, kvaid}@microsoft.com

More on Performance vs. Correctness

- Very similar to the latency-reliability tradeoff
  - Reliability is at the hardware component level
  - Correctness is at the program semantic level or hardware function level

- We have seen examples of the latency-reliability tradeoff before
  - See Lecture 12a: Low-Latency Memory
  - https://www.youtube.com/watch?v=mjiabRzGchI&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=12

Lecture on Latency-Reliability Tradeoff



https://www.youtube.com/watch?v=mjiabRzGchI&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=12

DRAM Latency PUFs

- Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu,
  "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices"
  *Proceedings of the 24th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Vienna, Austria, February 2018.
  [Lightning Talk Video]
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)]
  [Full Talk Lecture Video (28 minutes)]

The DRAM Latency PUF:
Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices

Jeremie S. Kim[†§] Minesh Patel[§] Hasan Hassan[§] Onur Mutlu[§†]
[†]Carnegie Mellon University [§]ETH Zürich

DRAM Latency True Random Number Generator

- Jeremie S. Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu,
"D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput"
*Proceedings of the 25th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Washington, DC, USA, February 2019.
[Slides (pptx) (pdf)]
[Full Talk Video (21 minutes)]
[Full Talk Lecture Video (27 minutes)]
Top Picks Honorable Mention by IEEE Micro.

D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput

Jeremie S. Kim[‡§] Minesh Patel[§] Hasan Hassan[§] Lois Orosa[§] Onur Mutlu[§‡]
[‡]Carnegie Mellon University [§]ETH Zürich

More on D-RaNGe

- Jeremie S. Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu,
  "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput"
  *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Washington, DC, USA, February 2019.
  [Slides (pptx) (pdf)]
  [Full Talk Video (21 minutes)]
  [Full Talk Lecture Video (27 minutes)]
  Top Picks Honorable Mention by IEEE Micro.

D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput

Jeremie S. Kim[‡§] Minesh Patel[§] Hasan Hassan[§] Lois Orosa[§] Onur Mutlu[§‡]
[‡]Carnegie Mellon University [§]ETH Zürich

In-DRAM True Random Number Generation

- Ataberk Olgun, Minesh Patel, A. Giray Yaglikci, Haocong Luo, Jeremie S. Kim, F. Nisa Bostanci, Nandita Vijaykumar, Oguz Ergin, and Onur Mutlu,
  "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips"
  Proceedings of the 48th International Symposium on Computer Architecture (***ISCA***), Virtual, June 2021.
  [Slides (pptx) (pdf)]
  [Short Talk Slides (pptx) (pdf)]
  [Talk Video (25 minutes)]
  [SAFARI Live Seminar Video (1 hr 26 mins)]

QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips

Ataberk Olgun[§†] Minesh Patel[§] A. Giray Yağlıkçı[§] Haocong Luo[§]
Jeremie S. Kim[§] F. Nisa Bostancı[§†] Nandita Vijaykumar[§⊙] Oğuz Ergin[†] Onur Mutlu[§]

[§]ETH Zürich *[†]TOBB University of Economics and Technology* *[⊙]University of Toronto*

Memory Ordering in Multiprocessors

Readings: Memory Consistency

- Required
  - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- Recommended
  - Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," ISCA 1990.
  - Gharachorloo et al., "Two Techniques to Enhance the Performance of Memory Consistency Models," ICPP 1991.
  - Ceze et al., "BulkSC: bulk enforcement of sequential consistency," ISCA 2007.

Memory Consistency vs. Cache Coherence

- Consistency is about ordering of all memory operations from different processors (i.e., to different memory locations)
  - Global ordering of accesses to *all* memory *locations*

- Coherence is about ordering of operations from different processors to the same memory location
  - Local ordering of accesses to *each* cache *block*

Difficulties of Multiprocessing

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs

Ordering of Operations

- Operations: A, B, C, D
  - In what order should the hardware execute (and report the results of) these operations?

- A contract between programmer and microarchitect
  - Specified by the ISA

- Preserving an "expected" (more accurately, "agreed upon") order simplifies programmer's life
  - Ease of debugging; ease of state recovery, exception handling

- Preserving an "expected" order usually makes the hardware designer's life difficult
  - Especially if the goal is to design a high performance processor: Recall load-store queues in out of order execution and their complexity

Memory Ordering in a Single Processor

- Specified by the von Neumann model
- Sequential order
  - Hardware executes the load and store operations in the order specified by the sequential program

- Out-of-order execution does not change the semantics
  - Hardware retires (reports to software the results of) the load and store operations in the order specified by the sequential program

- Advantages: 1) Architectural state is precise within an execution. 2) Architectural state is consistent across different runs of the program → Easier to debug programs
- Disadvantage: Preserving order adds overhead, reduces performance, increases complexity, reduces scalability

https://www.youtube.com/watch?v=-_JCpTWnEqc&list=PL5Q2soXY2Zi_uej3aY39YB5pfW4SJ7LlN&index=17

Memory Ordering in a Dataflow Processor

- A memory operation executes when its operands are ready

- Ordering specified only by data dependencies

- Two operations can be executed and retired in any order if they have no dependency

- Advantage: Lots of parallelism → high performance
- Disadvantage: No precise state (or ordering) semantics
  - Precise state is very hard to maintain (No specified order) → Very hard to debug
  - Order can change across runs of the same program → Very hard to debug

https://www.youtube.com/watch?v=pIAWMSNVt_8&list=PL5Q2soXY2Zi_uej3aY39YB5pfW4SJ7LlN&index=19

Memory Ordering in a MIMD Processor

- Each processor's memory operations are in sequential order with respect to the "thread" running on that processor (assume each processor obeys the von Neumann model)

- Multiple processors execute memory operations concurrently

- How does the memory see the order of operations from all processors?
  - In other words, what is the ordering of operations across different processors?

Why Does This Even Matter?

- Ease of debugging
  - It is useful to have the same execution done at different times to have the same order of execution → Repeatability
- Correctness
  - Can we have incorrect execution if the order of memory operations is different from the point of view of different processors?
- Performance and overhead
  - Enforcing a strict "sequential ordering" can make life harder for the hardware designer in implementing performance enhancement techniques (e.g., OoO execution, caches)

When Could Order Affect Correctness?

- When protecting shared data

Protecting Shared Data

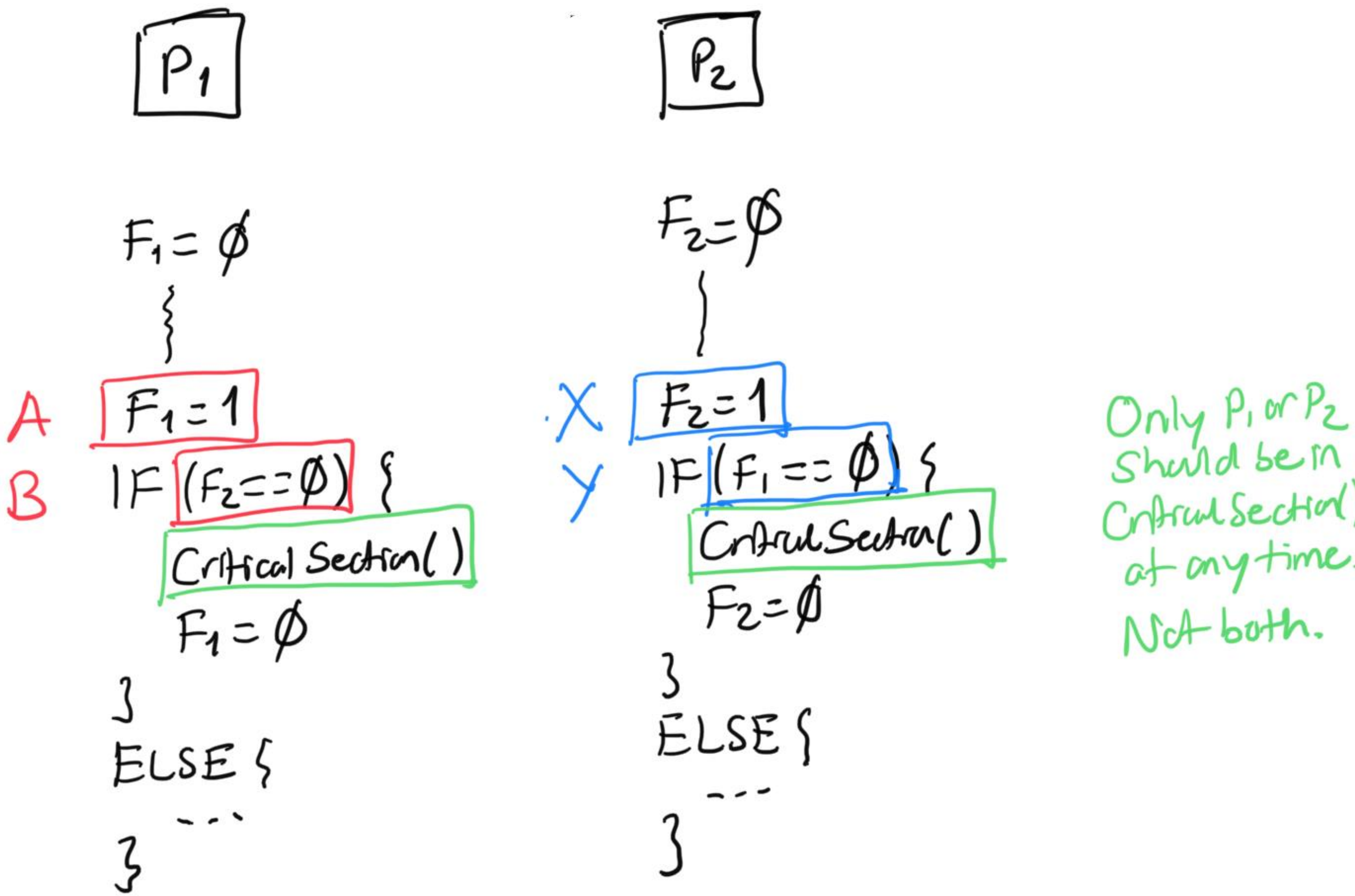
- Threads are not allowed to update shared data concurrently
  - For correctness purposes
- Accesses to shared data are encapsulated inside *critical sections or protected via synchronization constructs (locks, semaphores, condition variables)*
- Only one thread can execute a critical section at a given time
  - Mutual exclusion principle
- A multiprocessor should provide the *correct* execution of synchronization primitives to enable the programmer to protect shared data

Supporting Mutual Exclusion

- Programmer needs to make sure mutual exclusion (synchronization) is correctly implemented
  - We will assume this, i.e., threads are properly synchronized
  - But, correct parallel programming is an important topic
  - Reading: Dijkstra, "Cooperating Sequential Processes," 1965.
    - http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html
    - See Dekker's algorithm for mutual exclusion
- Programmer relies on hardware primitives to support correct synchronization
- If hardware primitives are not correct (or unpredictable), programmer's life is tough
- If hardware primitives are correct but not easy to reason about or use, programmer's life is still tough

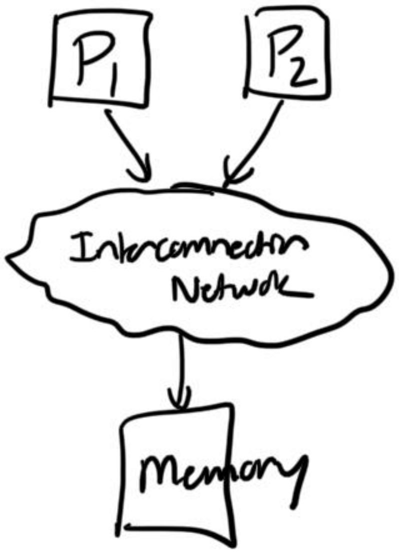
Protecting Shared Data

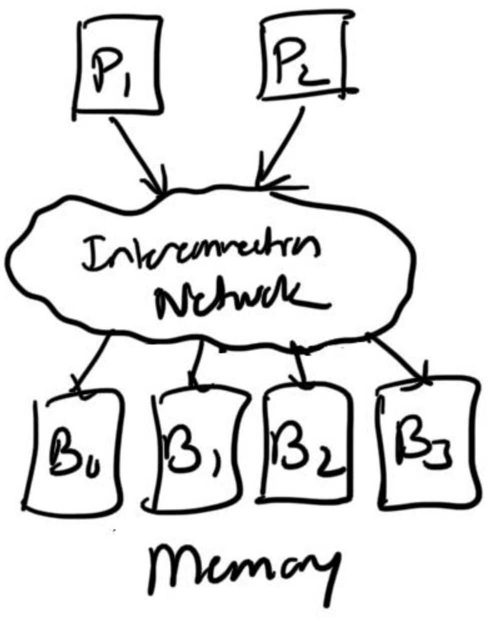
Assume P1 is in critical section.
Intuitively, it must have executed A,
which means F1 must be 1 (as A happens before B),
which means P2 should not enter the critical section.



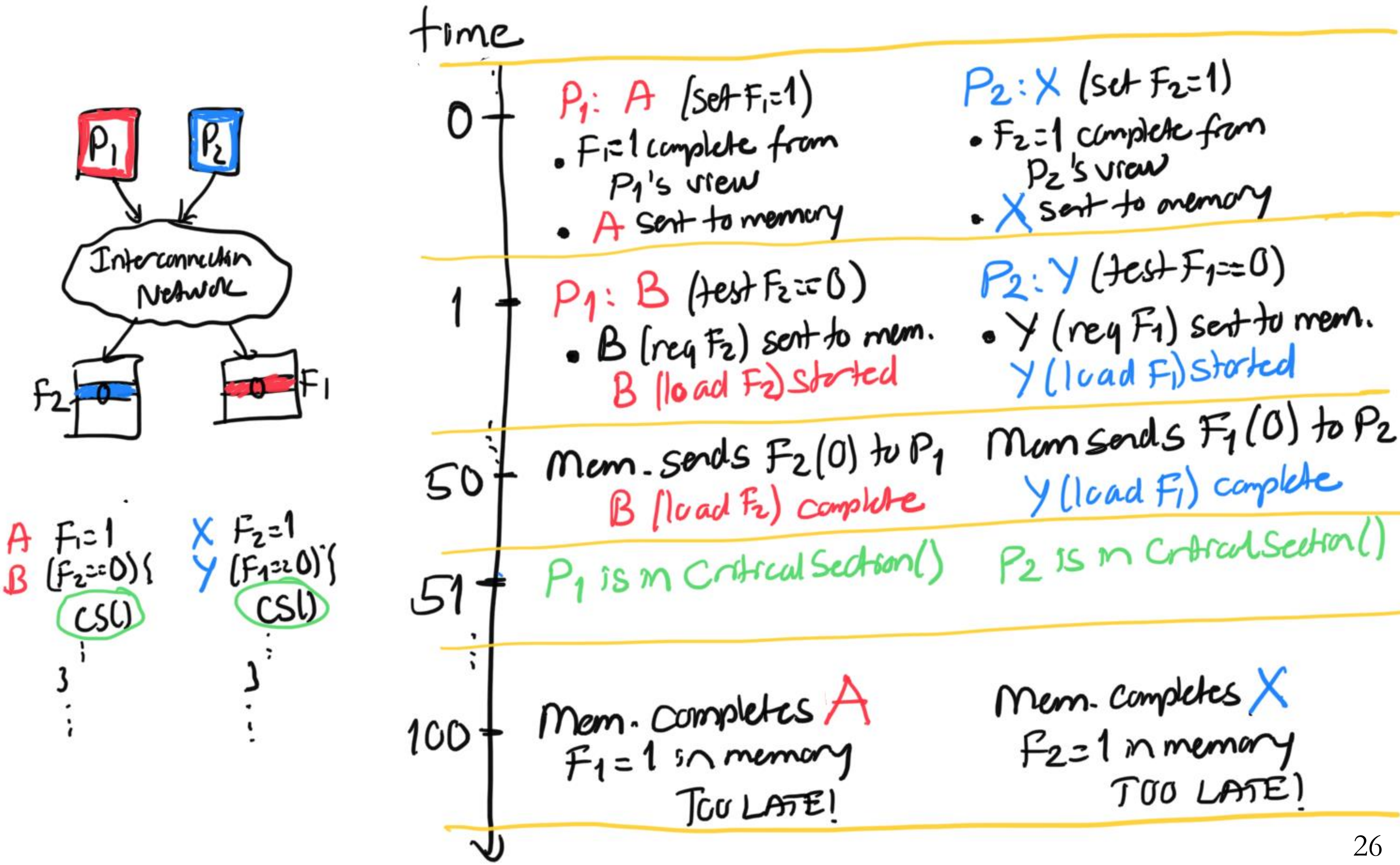
A Question

- Can the two processors be in the critical section at the same time given that they both obey the von Neumann model?
- Answer: yes



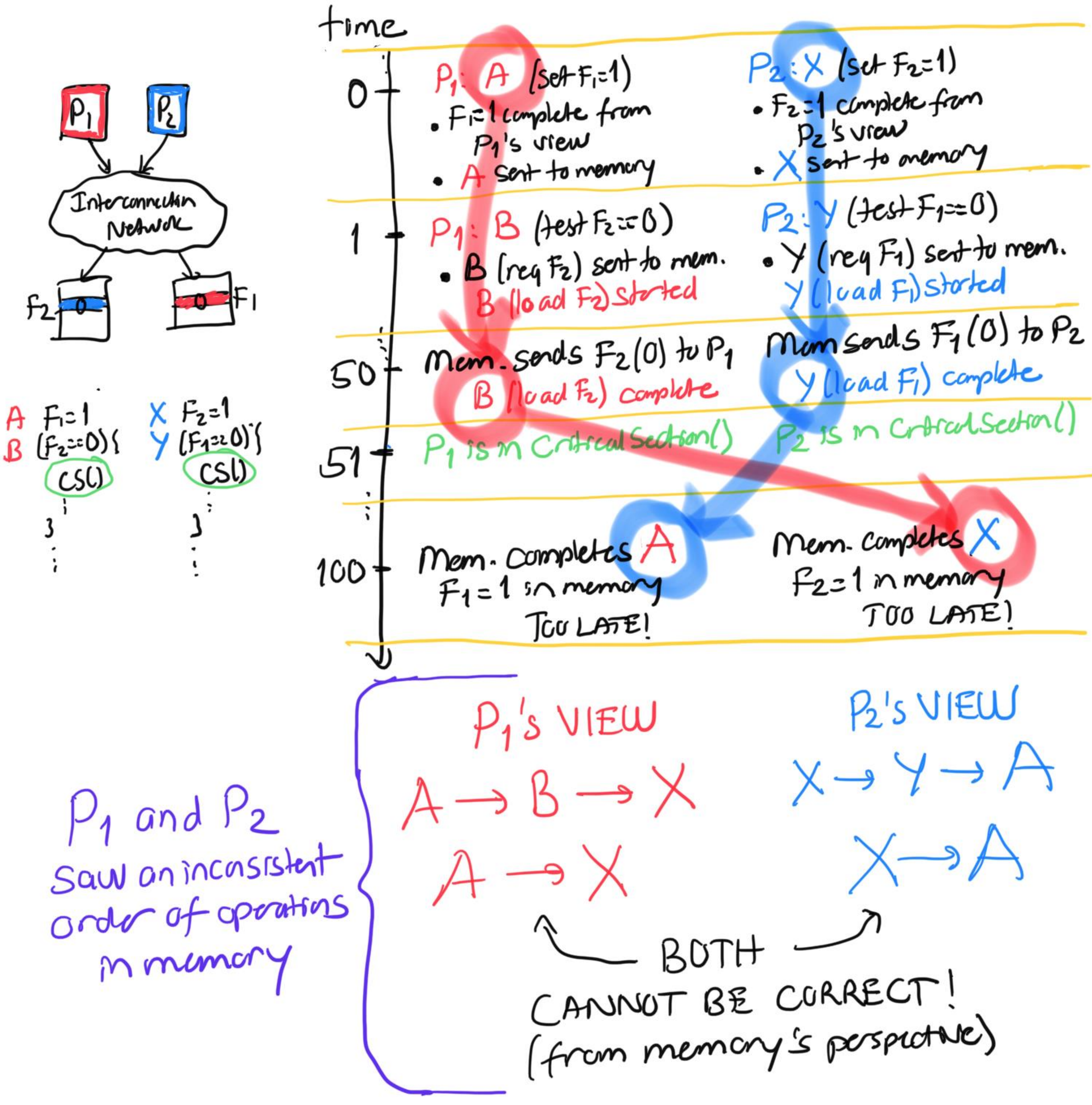


Incorrect Result: Both Processors in Critical Section



What Happened?

Let's Examine Each Processor's View



For P1:
A appeared to happen before X

For P2:
X appeared to happen before A

The Problem

- The two processors did **NOT** see the same order of operations to memory
- The "happened before" relationship between multiple updates to memory was inconsistent between the two processors' points of view
- As a result, each processor thought the other was **not** in the critical section

How Can We Solve The Problem?

- Idea: Sequential consistency

- All processors see the same order of operations to memory
- i.e., all memory operations happen in an order (called the global total order) that is consistent across all processors

- Assumption: within this global order, each processor’s operations appear in sequential order with respect to its own operations.

Sequential Consistency

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- A multiprocessor system is sequentially consistent if:
  - the result of any execution is the same as if the operations of all the processors were executed in some sequential order

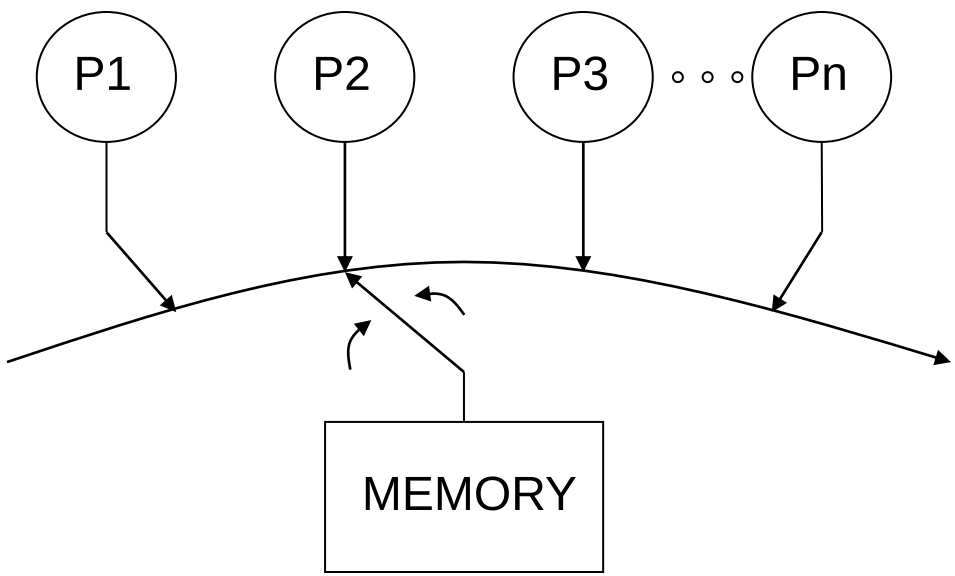
AND

  - the operations of each individual processor appear in this sequence in the order specified by its program

- This is a memory ordering model, or memory model
  - Specified by the ISA

Programmer’s Abstraction

- Memory is a switch that services one load or store at a time from any processor
- All processors see the currently serviced load or store at the same time
- Each processor’s operations are serviced in program order



Sequentially Consistent Operation Orders

- Potential correct global orders (all are correct):

- A B X Y
- A X B Y
- A X Y B
- X A B Y
- X A Y B
- X Y A B

- Which order (interleaving) is observed depends on implementation and dynamic latencies

Consequences of Sequential Consistency

- Corollaries

1. Within the same execution, all processors see the same global order of operations to memory
   → No correctness issue
   → Satisfies the "happened before" intuition

2. Across different executions, different global orders can be observed (each of which is sequentially consistent)
   → Debugging is still difficult (as order changes across runs)

Lamport Paper from 1979

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Index Terms—**Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the

Issues with Sequential Consistency?

- Nice abstraction for programming, but two issues:
  - Too conservative ordering requirements
  - Limits the aggressiveness of performance enhancement techniques

- Is the total global order requirement too strong?
  - Do we need a global order across all operations and all processors?
  - How about a global order only across all stores?
    - Total store order memory model; unique store order model
  - How about enforcing a global order only at the boundaries of synchronization?
    - Relaxed memory models
    - Acquire-release consistency model

Issues with Sequential Consistency?

- Performance enhancement techniques that could make SC implementation difficult

- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores → makes it difficult for all processors to see the same global order of all memory operations

- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors → makes it difficult for all processors to see the same global order of all memory operations

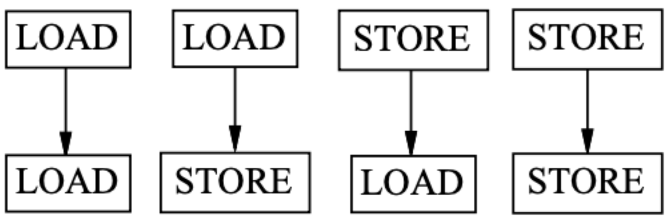
Weaker Memory Consistency

- The ordering of operations is important when the order affects operations on shared data → i.e., when processors need to synchronize to execute a "program region"

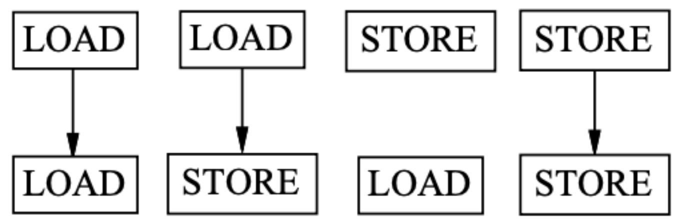
- Weak consistency
  - Idea: Programmer specifies regions in which memory operations do not need to be ordered
  - "Memory fence" instructions delineate those regions
    - All memory operations before a fence must complete before fence is executed
    - All memory operations after the fence must wait for the fence to complete
    - Fences complete in program order
  - All synchronization operations act like a fence

Examples of Weak Consistency Models

- Gharachorloo et al., "Two Techniques to Enhance the Performance of Memory Consistency Models," ICPP 1991.

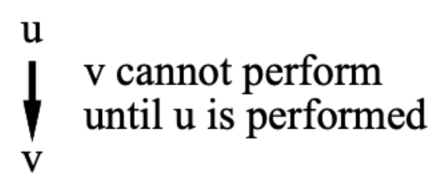


Sequential Consistency (SC)



Processor Consistency (PC)

A consistency model imposes restrictions on the order of shared memory accesses initiated by each process. The strictest model, originally proposed by Lamport [15], is sequential consistency (SC). Sequential consistency requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Processor consistency (PC) was proposed by Goodman [9] to relax some of the restrictions imposed by sequential consistency. Processor consistency requires that writes issued from a processor may not be observed in any order other than that in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. Sufficient constraints to satisfy processor consistency are specified formally in [8].



Examples of Weak Consistency Models

- Gharachorloo et al., "Two Techniques to Enhance the Performance of Memory Consistency Models," ICPP 1991.

A more relaxed consistency model can be derived by relating memory request ordering to synchronization points in the program. The weak consistency model (WC) proposed by Dubois et al. [4, 5] is based on the above idea and guarantees a consistent view of memory only at synchronization points. As an example, consider a process updating a data structure within a critical section. Under SC, every access within the critical section is delayed until the previous access completes. But such delays are unnecessary if the programmer has already made sure that no other process can rely on the data structure to be consistent until the critical section is exited. Weak consistency exploits this by allowing accesses within the critical section to be pipelined. Correctness is achieved by guaranteeing that all previous accesses are performed before entering or exiting each critical section.

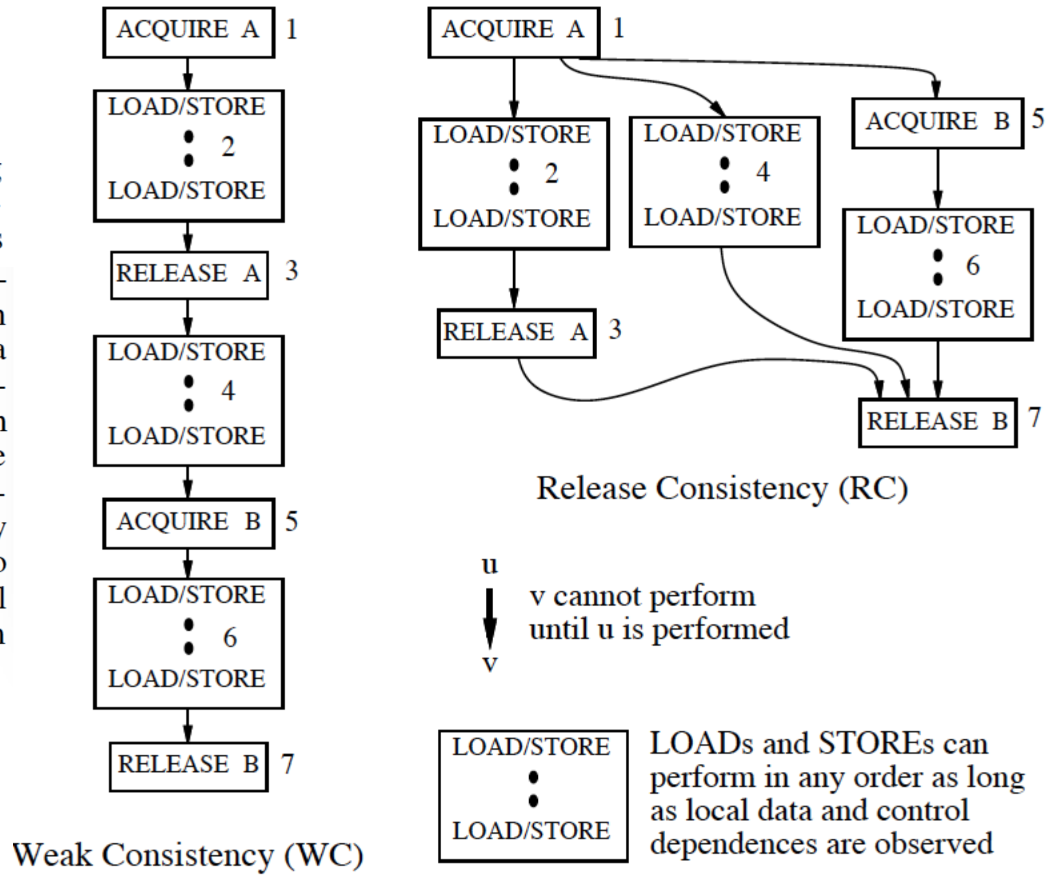


Figure 1: Ordering restrictions on memory accesses.

More on Weak Consistency

- Dubois et al., "Memory Access Buffering in Multiprocessors," ISCA 1986.

- Dubois et al., "Memory Access Dependencies in Shared-Memory Multiprocessors," IEEE TSE 1990.

Examples of Weak Consistency Models

- Gharachorloo et al., "Two Techniques to Enhance the Performance of Memory Consistency Models," ICPP 1991.

Release consistency (RC) [8] is an extension of weak consistency that exploits further information about synchronization by classifying them into acquire and release accesses. An *acquire* synchronization access (e.g., a lock operation or a process spinning for a flag to be set) is performed to gain access to a set of shared locations. A *release* synchronization access (e.g., an unlock operation or a process setting a flag) grants this permission. An acquire is accomplished by reading a shared location until an appropriate value is read. Thus, an acquire is always associated with a read synchronization access (see [8] for discussion of read-modify-write accesses). Similarly, a release is always associated with a write synchronization access. In contrast to WC, RC does not require accesses following a release to be delayed for the release to complete; the purpose of the release is to signal that previous accesses are complete, and it does not have anything to say about the ordering of the accesses following it. Similarly, RC does not require an acquire to be delayed for its previous accesses. The data-race-free-0 (DRF0) [2] model

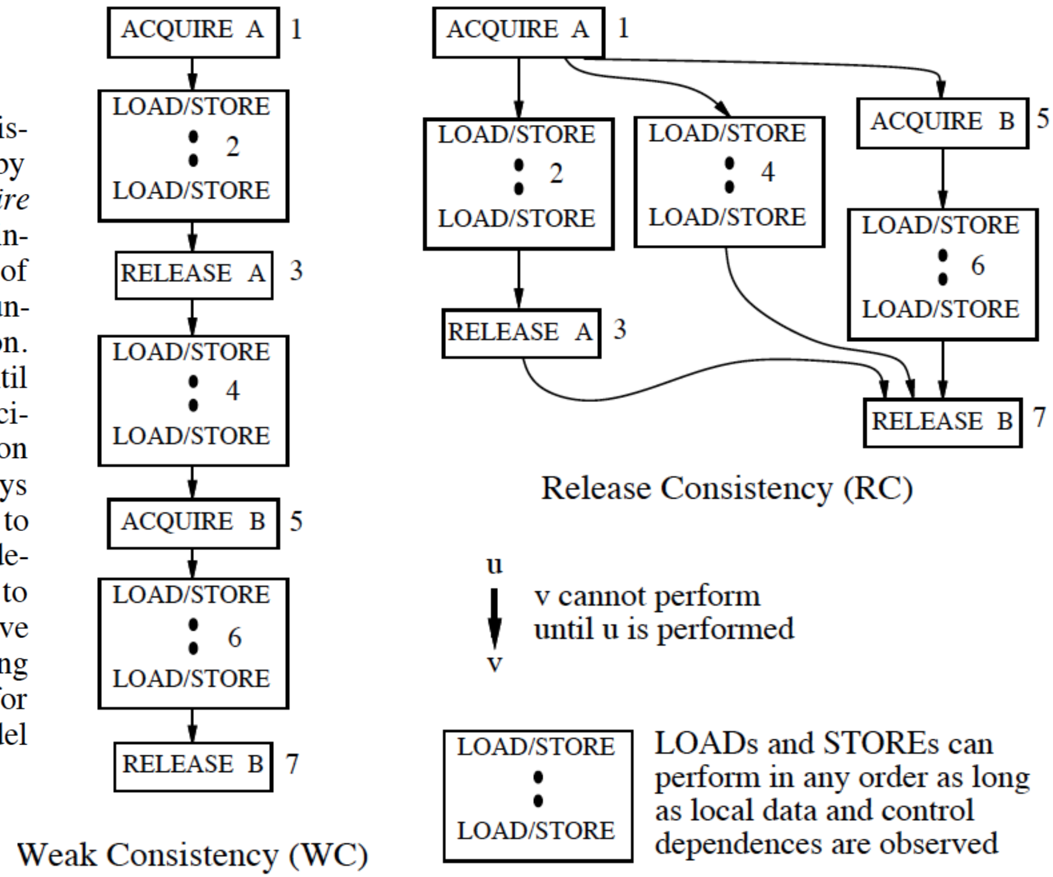


Figure 1: Ordering restrictions on memory accesses.

More on Release Consistency

- Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," ISCA 1990.

Tradeoffs: Weaker Consistency

- Advantage
  - No need to guarantee a very strict order of memory operations
    - → Enables the hardware implementation of performance enhancement techniques to be simpler
    - → Can be higher performance than stricter ordering

- Disadvantage
  - More burden on the programmer or software (need to get the "fences" correct)

- Another example of the programmer-microarchitect tradeoff

More on Weak Consistency Models

- Gharachorloo et al., "Two Techniques to Enhance the Performance of Memory Consistency Models," ICPP 1991.

Abstract

The memory consistency model supported by a multiprocessor directly affects its performance. Thus, several attempts have been made to relax the consistency models to allow for more buffering and pipelining of memory accesses. Unfortunately, the potential increase in performance afforded by relaxing the consistency model is accompanied by a more complex programming model. This paper introduces two general implementation techniques that provide higher performance for all the models. The first technique involves *prefetching* values for accesses that are delayed due to consistency model constraints. The second technique employs *speculative execution* to allow the processor to proceed even though the consistency model requires the memory accesses to be delayed. When combined, the above techniques alleviate the limitations imposed by a consistency model on buffering and pipelining of memory accesses, thus significantly reducing the impact of the memory consistency model on performance.

Two Example Questions

Example Question I

- Question 4 in
  - http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=final.pdf

4. Sequential Consistency [30 points]

Two threads (A and B) are concurrently running on a dual-core processor that implements a *sequentially consistent* memory model. Assume that the value at address 0x1000 is initialized to 0.

Thread A

```
X1: st 0x1, (0x1000)
X2: ld $r1, (0x1000)
X3: st 0x2, (0x1000)
X4: ld $r2, (0x1000)
```

Thread B

```
Y1: st 0x3, (0x1000)
Y2: ld $r3, (0x1000)
Y3: st 0x4, (0x1000)
Y4: ld $r4, (0x1000)
```

(a) List all possible values that can be stored in `$r3` after both threads have finished executing.

Example Question I (Continued)

(b) After both threads have finished executing, you find that (`$r1`, `$r2`, `$r3`, `$r4`) = (1, 2, 3, 4). How many different *instruction interleavings* of the two threads produce this result?

(c) What is the total number of all possible instruction interleavings? You need not expand factorials.

(d) On a *non-sequentially consistent* processor, is the total number of all possible instruction interleavings less than, equal to, or greater than your answer to question (c)?

Example Question II

- Question 8 in
  - https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=final-fs19.pdf

8 Memory Consistency [35 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) [5 points] Do you find something that could be wrong in the C code segments? Explain your answer.

Example Question II (Continued)

(b) [10 points] 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.

(c) [5 points] 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.

(d) [5 points] What could be possible final values of `b` in the SC processor, after both threads finish execution? Explain your answer. Provide all possible values.

(e) [10 points] 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.

Computer Architecture
Lecture 19b: Memory Ordering (Memory Consistency)

Prof. Onur Mutlu
ETH Zürich
Fall 2021
2 December 2021