Cache Coherence and Memory Consistency

Shuhao Zhang

Nanyang Technological University

shuhao.zhang@ntu.edu.sg

June 3, 2024

"...when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced. ...

I would be panicked if I were in industry."

John Hennessy, President, Stanford University, 2007

Parallel Computing is Challenging

Parallel Programming Challenges:

- **Finding enough parallelism (Amdahl's Law!)**
- **Granularity**
- **Locality**
- **Load balance**
- **Coordination and synchronization**
- Correctness/Debugging
- Performance modelling / Monitoring

Those are what the foster's design methodology emphsize

Parallel Computing is Challenging

Parallel Programming Challenges:

- Finding enough parallelism (Amdahl's Law!)
- Granularity
- Locality
- Load balance
- Coordination and synchronization
- **Correctness/Debugging!** ;)
- Performance modelling / Monitoring

How to make sure our multi-threading program is a correct one?

A Joke

Some people, when confronted with a problem, think, 'I know, I'll use threads' - and then two they hav erpoblesms.

What is the Problem?

1. A sequential program consisted of a sequence of instructions (and a memory), where each instruction executed one after the other (to modify the memory, etc.).
2. The sequential paradigm has the following two characteristics: the textual order of statements specifies their order of execution[1]; successive statements must be executed without any overlap (in time) with one another.

Remark

Both are not true in parallel computing.

[1] there are exceptions.

What is the Problem?

- Threads cooperate during parallel computing[2]
- Share resources, access shared data structures
- Scheduling is (by default) not under program control, but by who?
- Threads interleave executions arbitrarily and at different rates

[2] Discuss in terms of threads, but also applies to processes.

Common Correctness Issues

- Visibility issues
- Race conditions
- Execution ordering
- Deadlocks

Outline

- This week: visibility issues
  - Cache coherence and memory consistency
- Next Week: other issues

Why Cache Matters?

How does cache matters to parallel computing? It matters in terms of both correctness and efficiency of parallel program.

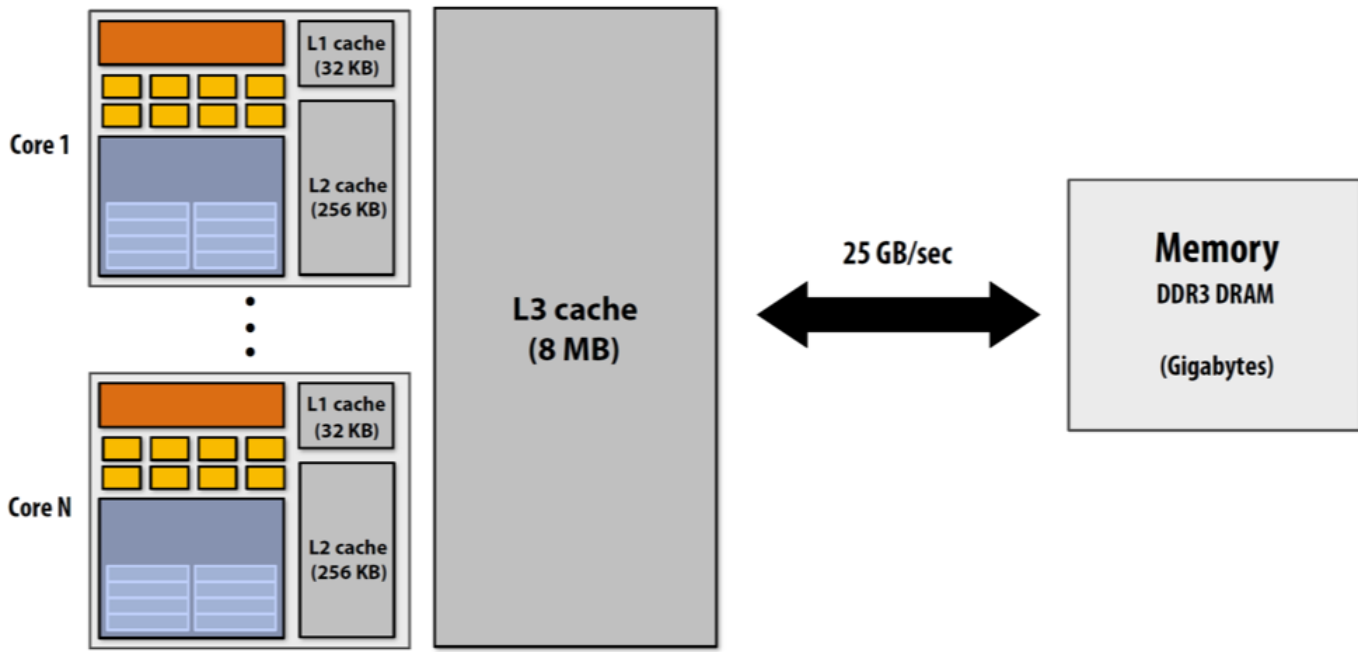
- efficiency: locality principle
- correctness: visibility issue

Notes

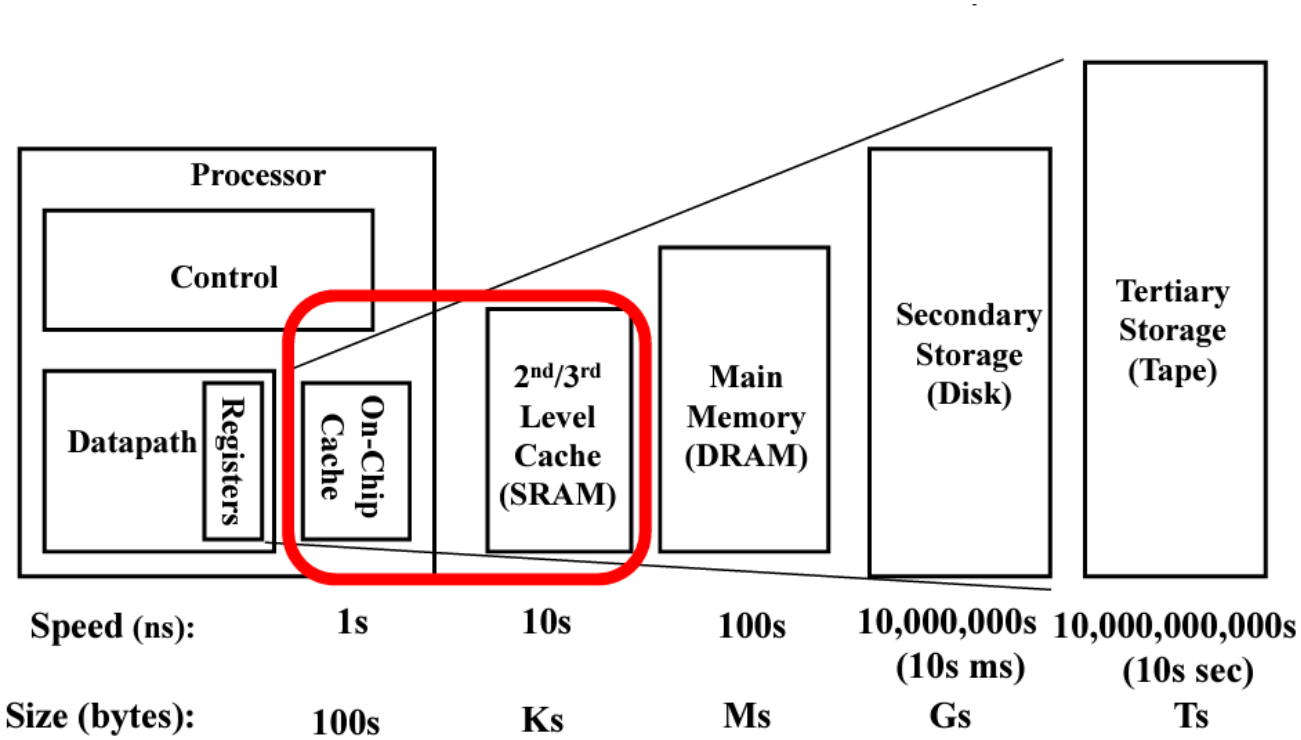
Let's begin with 'efficiency' aspect.

Why do Modern Processors have Cache?

- Processors run efficiently when data is resident in caches
  - Caches reduce memory access latency
  - Caches provide high bandwidth data transfer to CPU



Why do Modern Processors have Cache?



Cache Properties

- **Cache size**: larger cache increases access time (because of increased addressing complexity) but reduces cache misses
- **Block size**: data is transferred between main memory and cache in blocks of a fixed length
  - Larger blocks reduces the number of blocks but replacement costs more → block size should be small
  - But larger block increase the chance of spatial locality cache hit → block size should be large

Remark

Typical sizes for L1 cache blocks are 4 or 8 memory words

Principle of Locality

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves
  - **Spatial locality**: Items with nearby addresses tend to be referenced close together in time
  - **Temporal locality**: Recently referenced items are likely to be referenced in the near future

Locality Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

From "Data"'s view:

- Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
- Reference sum each iteration: **Temporal locality**

From "Instruction"'s view:

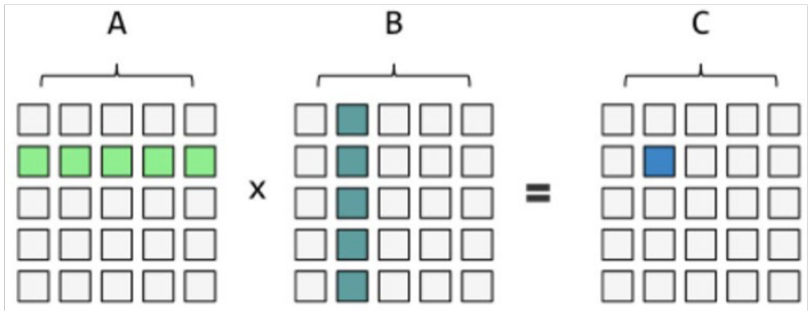
- Reference instructions in sequence: **Spatial locality**
- Cycle through loop repeatedly: **Temporal locality**

Sources of locality

```
for (i=0; i<N; i++) {
    A[i] = B[i] + C[i] * a;
}
```

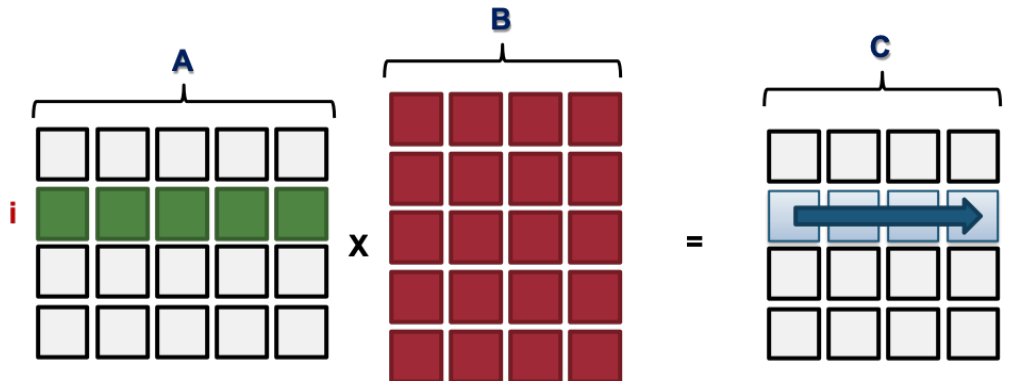
- Temporal locality:
  - Code within a loop
  - Same instructions fetched repeatedly
- Spatial locality:
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks (contiguous bytes)

Matrix Multiplication




```
for i ← 0 to n-1
    for j ← 0 to n-1
        c[i, j] ← 0
        for k ← 0 to n-1
            c[i, j] ← c[i, j] + a[i, k] x b[k, j]
```

One Iteration



- Read:
  - Row i^{th} of matrix A
  - Entire matrix B
- Write:
  - Row i^{th} of matrix C

Remark

What are the potential cache related performance problems?

Matrix Multiplication and Cache

- Matrix row-column ordering:
  - Row-major vs Column-major
- Size of matrix: How large is the square matrix that can be stored entirely in a 256KB cache?
- You can assume double-precision floating point numbers (i.e., 8 bytes).

Why Cache Matters?

Cache is much faster than memory, and locality is a key performance factor.

Notes

Let's now come back to the 'correctness' aspect.

A Coherent Memory System

- Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item.
- This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs.
  - The first aspect, called **coherence**, defines what values can be returned by a read.
  - The second aspect, called **consistency**, determines when a written value will be returned by a read.

Properties of a coherent memory system

- A coherent memory system is coherent if the following hold good:
  - **Program Order** property:
    - Given the sequence: 1) P write to X; 2) No write to X; 3) P read from X
    - P should get the value written in 1)
  - **Write Propagation** property:
    - Given the sequence: 1) P1 write to x; 2) No further write to x; 3) P2 read from x
    - P2 should read value written by P1
    - That is, writes become visible to other processors
  - **Write Serialization** property:
    - Given the sequence: 1) Write V1 to X (by any processor); 2) Write V2 to X (by any processor)
    - Processors can never read X as V2, then later as V1
    - All writes to a location (from the same or different processors) are seen in the same *order* by all processors

Formal Definition of Cache Coherence

Cache Coherence: ensures that each processor has consistent view of memory through its local cache

- All writes to **SAME memory location** (address) should be seen by all processors in the same order
- Writes to an address by one processor will eventually be observed by other processors – **but does not specify when**

Implication of Cache Coherence for Multi-threading Programming

- Intuitively, reading value at an address should return the last value written at that address by any processor
- Multiprocessor: a single share address space leads to a memory coherence problem because there is
  - Global storage space (main memory)
  - Per-processor local storage (per-processor caches)

Cache Lines

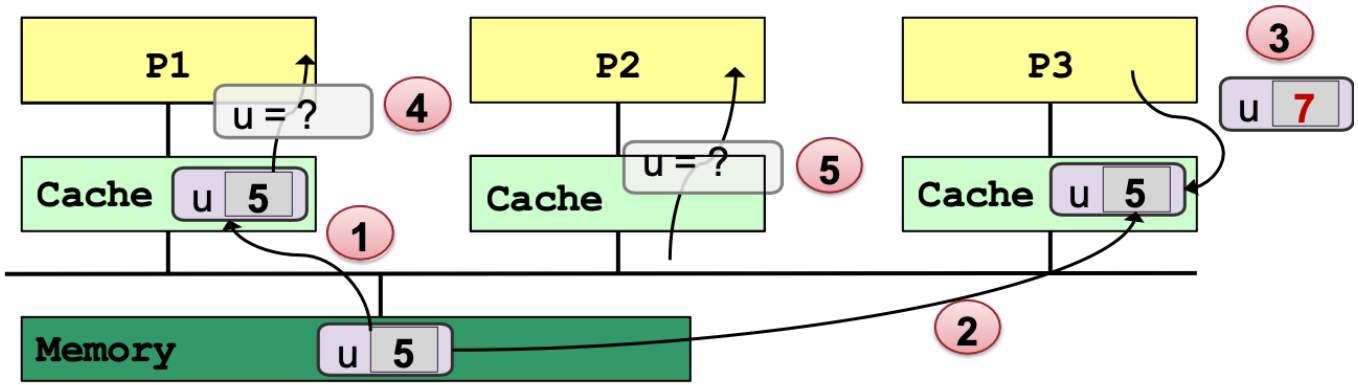
- When the CPU caches are reading data from lower level caches or main RAM (e.g. L1 from L2, L2 from L3 and L3 from main RAM), they read a **cache line**.
- A cache line typically consists of 64 bytes. Thus, the caches read 64 bytes at a time from lower level caches or main RAM.

Cache Line Invalidation

- When a CPU writes to memory address in a cache line, typically because the CPU is writing to a variable, the cache line becomes dirty.
- The cache line then needs to be synchronized to other CPUs that also have that cache line in their CPU caches. The same cache line stored in the other CPU caches thus becomes invalid - they need to be refreshed, in other words.
- **Cache refreshing** after cache invalidation can happen either via cache coherence mechanisms, or by reloading the cache line from main RAM.
- The CPU is not allowed to access that cache line until it has been refreshed.

Cache Coherence Problem

- Multiple copies of the same data exists on different caches
- Local update by processor → Other processors may still see the unchanged data



Maintaining Cache Coherence

- Cache coherence can be maintained by:
  - Software based solution
    - OS + Compiler + Hardware aided solution
    - E.g., OS uses page-fault mechanism to propagate writes
  - Hardware based solution
    - Most common on multiprocessor system
    - Known as *cache coherence protocols*

Software-based Cache Coherence Maintenance: Write Policy

- Write-through: write access is immediately transferred to main memory
  - Advantage: always get the newest value of a memory block
  - Disadvantages: slow down due to memory accesses
    - A simple trick to improve: use a write buffer – **buffer write policy**
- Write-back: write operation is performed only in the cache
  - write is performed to the main memory when the cache block is replaced
  - uses a dirty bit to indicates whether the cache is still valid
  - Advantages: less write operations
  - Disadvantages: memory may contain invalid entries; onus

Remark

By default, python Java/C program follows **write-back** strategy

FYI: Write-back Cache

Example: processor executes int x = 1;

1. Processor performs write to address that "misses" in cache
2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory
3. Cache loads line from memory ("allocates line in cache")
4. Whole cache line is fetched
5. Cache line is marked as dirty

Line state	Tag	Data (64 bytes)

FYI: Hardware-based Cache Coherence Maintenance

Two major categories:

- Snooping Based
  - No centralized directory
  - Each cache keeps track of the sharing status
  - Cache monitors or snoop on the bus
    - to update the status of cache line
    - takes appropriate action
  - Most common protocol used in architectures with a bus
- Directory Based
  - Sharing status is kept in a centralized location
  - Commonly used with NUMA architectures

Cache Coherence Implications

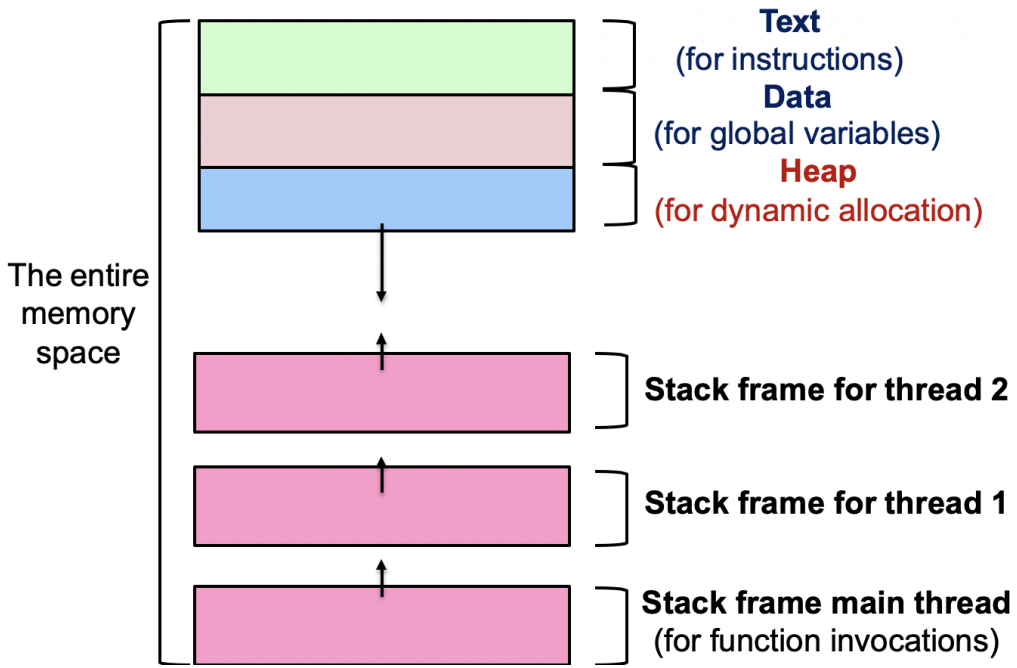
- Overhead in shared address space:
  - CC appears as increased memory latency in multiprocessor
  - CC lowers the hit rate in cache
- Two Common Cache Coherence Problems:
  - Visibility of Global Variable
  - False sharing

Case Study: “Global" Variables

- Define static variable of a class, and all threads of the class can “see" (read and write) to that variable.
- FYI: There’s no official “interrupt" API in C/C++, we can use global variable to simulate it. (How?)

Visibility of Variables

- Global variables of a program and all dynamically allocated data objects can be accessed by any thread of this process
- Each thread has a private runtime stack for function stack frames
- Runtime stack of a thread exists iff the thread is active



Java Memory Model

- The Java Memory Model proposes a weaker guarantee called *Data-Race Free Guarantee*:
  - A program is said to be correctly synchronized or data-race-free *iff* all sequentially consistent executions of the program are free of data races.

The **volatile** Keyword

Force the write to a volatile variable to be dump to main memory and ensures subsequent read to that variable are executed after write:

- It enforces the *cache coherence* in JVM (even if the HW does not support it)
- and surely setup the *happen-before* relation as a result ;)

Caution

Self-study: *volatile* keyword in C/C++ stands for different purposes, use $std :: atomic < T >$ instead. `https://en.wikipedia.org/wiki/Volatile_(computer_programming)`

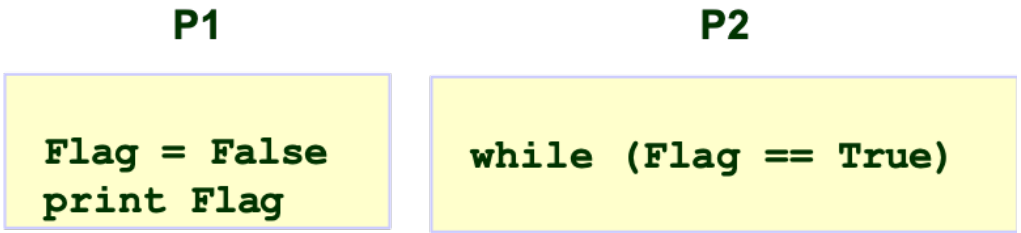
KeepRunning Experiment

```
public class keepRunningExperiment extends Thread {
    volatile boolean keepRunning = true;
    public static void main(String[] args) throws InterruptedException {
        Visibility t = new Visibility();
        t.start();
        Thread.sleep(1000);
        t.keepRunning = false;
        System.out.println(System.currentTimeMillis() + ": keepRunning
            is false");
    }
    public void run(){
        while(keepRunning){
        }
    }
}
```

Remark

What will happen here? What if we remove the ‘volatile‘ keyword?

KeepRunning Experiment (Cont'd)

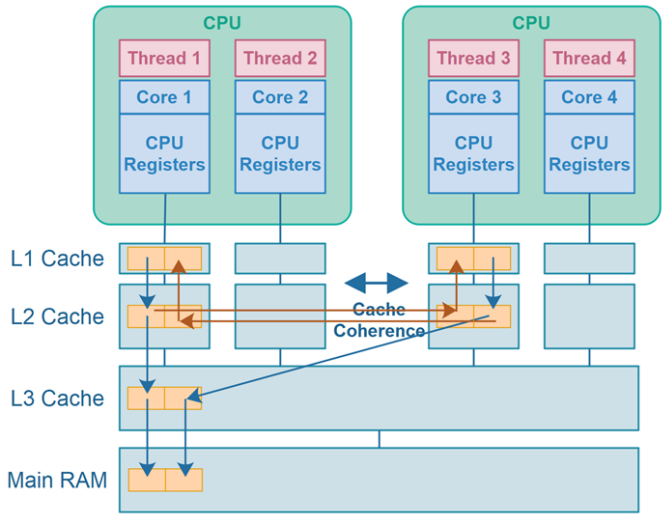


- Data Race: Two accesses x and y form a data race in an execution of a program if they are from different threads, they conflict, and they are not ordered by happens-before → this program is **NOT** data-race-free
- And, it is your responsibility to ensure its correctness (not by JVM!)

Recall: Cache Lines

- A cache line typically consists of 64 bytes. Thus, the caches read 64 bytes at a time from lower level caches or main RAM.
- A single cache line will often store more than one variable.
  - If the same CPU needs to access more of the variables stored within the same cache line - this is an advantage.
  - If multiple CPUs need to access the variables stored within the same cache line, **false sharing** can occur.

Case Study: False sharing problem



The diagram shows two threads running on different CPUs which write to different variables - with the variables being stored within the same CPU cache line - causing false sharing.

False Sharing:

- 2 processors write to different addresses, but..
- The addresses map to the same cache line

Fixing False Sharing

- The way to fix a false sharing problem is to design your code so that different variables used by different threads do not end up being stored within the same CPU cache line.
- Exactly how you do that depends on your concrete code, but storing the variables in different objects is one way to do so - as the example in the previous section showed.
- Another way is to use @Contented keyword since Java 8.

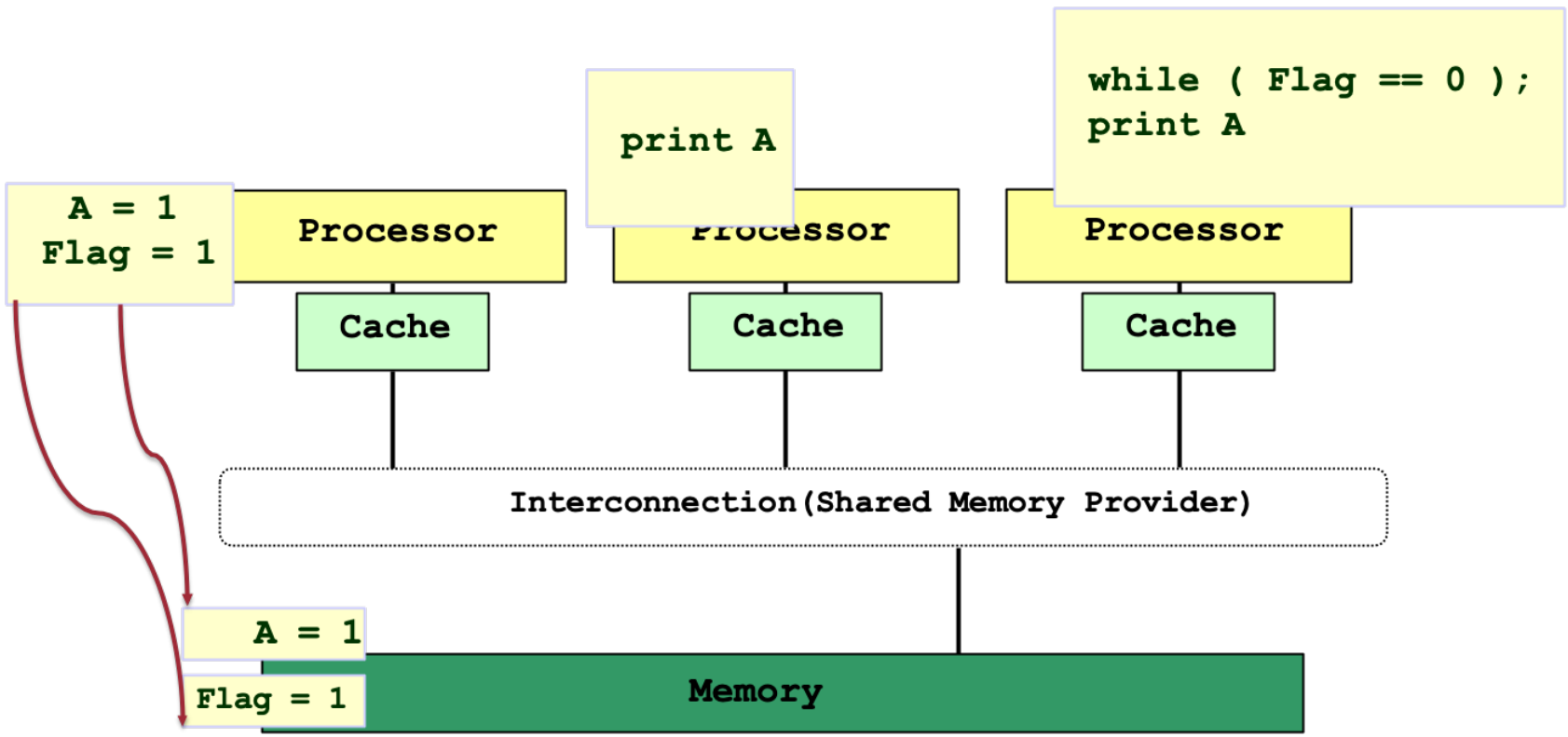
Memory Consistency vs. Cache Coherence

- **[Recap] Cache Coherence**: ensures that each processor has consistent view of memory through its local cache
  - All writes to **SAME memory location** (address) should be seen by all processors in the same order
  - Writes to an address by one processor will eventually be observed by other processors – **but does not specify when**
- **Memory consistency** ensures
  - Constraints on the order in which memory operations can appear to execute – **when the operations are seen by other processors?**
  - For **DIFFERENT memory locations**
- The consistency model is then used:
  - By programmers to reason about correctness and program behavior
  - By system designers to decide the reordering possible by hardware and compiler

Warm up: Uniprocessor

- In uniprocessor, we assume:
  - Memory operations are performend in program order
  - Memory operations are **atomic**
- But, actually:
  - Operations maybe reordered to improve performance
  - Constraint on reordering: must respect dependence:
    - control dependence must be respected
    - data dependence must be respected: in particular, loads/stores to a given memory address must be executed in program order (but not constrained for different memory locations)
    - Read maybe out of order w.r.t write for different memory locations
    - Write may not be reflected in main memory immediately

Memory Consistency Problem



Memory Consistency Models

- **Memory consistency model** is sort of a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of reading, writing, or updating memory will be predictable.

Types of Memory Consistency Models

- issue and view methods:
  - **Issue**: issue method describes the restrictions that define how a process can issue operations.
  - **View:** View method which defines the order of operations visible to processes.
- For example, a consistency model can define that a process is not allowed to issue an operation until all previously issued operations are completed.
- One consistency model can be considered stronger than another if it requires all conditions of that model and more.

Strict consistency

A shared-memory system is said to support the strict consistency model if

- the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations
- all writes instantaneously become visible to all processes

Remark

Only applicable for uniprocessor system as it assumes concurrent writes to be impossible.

Sequential Consistency Model (SC) – Lamport 1976

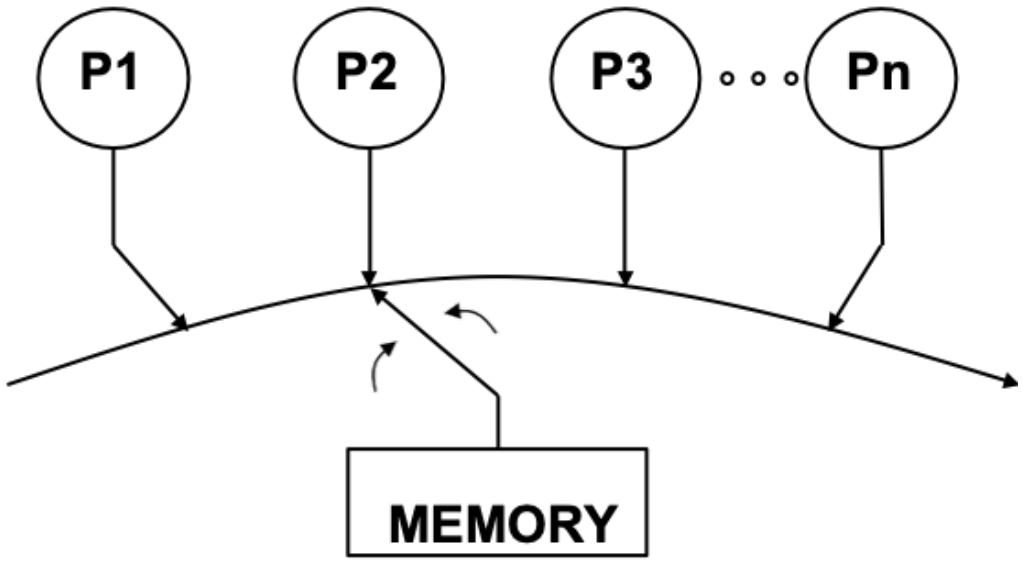
A shared-memory system is said to support the sequential consistency model if

- Every processor issues its memory operations in program order
- all processes see the *same order* of all memory access operations on the shared memory
- it does not specify the exact order in which the memory access operations are interleaved
- effect of each memory operation must be visible to all processors before the next memory operation on any processor

Remark

It is an intuitive extension of strict consistency model but can result in loss of performance

Sequential Consistency: Illustration



- Program order: memory ordering has to follow the individual order in each thread
- Write-atomicity: there can be any interleaving of such sequential segments - but a single total order of all memory operations
- **As if** only one memory operation at any point in time

Motivation for Relaxed Consistency (RC)

- Hide latencies!
  - To gain performance
  - Specifically, hiding memory latency: overlap memory access operations with other operations when they are independent
- How?
  - Overlap memory access operations with other operations when they are independent

Relaxed Consistency

- Relaxed Consistency – relax the ordering of memory operations if data dependencies allow
  - If two operations access the same memory location:
    - R → W: anti-dependence (WAR)
    - W → W: output dependence (WAW)
    - W → R: flow dependence (RAW)
- Relaxed memory consistency models allow certain orderings to be violated

Warning

Program order must be preserved: memory ordering has to follow the individual order in each thread/core

Relaxed Consistency: Idea

Processor	P_1	P_2	P_3
Program	(1) $x_1 = 1$; (2) print (x_2)	(3) $x_2 = 1$; (4) print (x_1)	(5) $x_3 = 1$; (6) print (x_3)

SC:

- What are the valid operation orders?
- e.g., (1)-(2)-(5)-(6)-(3)-(4)

Example of RC:

- relax W → R
- Possible with RC:
  - (3)-(4) or (4)-(3)
  - (1)-(2) or (2)-(1)
  - (5)-(6)
  - not: (6)-(5) → program order must be preserved

Relaxed Memory Consistency Models

- Memory models resulted from relaxation of SC's requirements
- Program order relaxation:
  - Write → Read
  - Write → Write
  - Read → Read or Write
- All models provide overriding mechanism to allow a programmer to intervene

Write-to-Read Order Relaxing

- Key Idea:
  - Allow a read on processor **P** to be reordered w.r.t. to the previous write of the same processor
    - so as to hide the write latency
    - different timing of the return of the read further defines different consistency models: Total Store Ordering (TSO); Processor Consistency (PC)
- TSO: Return the value written by **P** earlier without waiting for it to be serialized
- PC: Return the value of any write (even from another processor) before the write is propagated or serialized

Remark

TSO appears to match the memory consistency model of the widely used x86 architecture

Example 1

P1

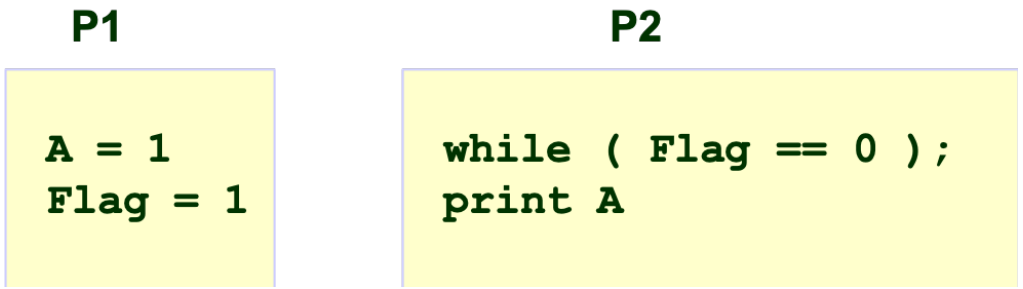
```
A = 1
Flag = 1
```

P2

```
while ( Flag == 0 );
print A
```

- A = Flag = 0 initially
- Can A = 0 be printed under the models of SC/TSO/PC?

Example 1 (cont'd)



- A = Flag = 0 initially
- Can A = 0 be printed under the models of SC/TSO/PC?
- *Impossible.*

Example 2

- A = B = 0 initially
- Can A = 0; B = 0 be printed under the models of SC/TSO/PC?

Example 2 (cont'd)

- A = B = 0 initially
- Can A = 0; B = 0 be printed under the models of SC/TSO/PC?
- *Impossible for SC, but possible for TSO/PC*
  - because TSO/PC can violate $W \rightarrow R$

Write-to-Write Order Relaxing

- Key Idea:
  - Writes can bypass earlier writes (to different locations) in write buffer
  - Allow write miss to overlap and hide latency
- Example Model:
  - Partial Store Ordering (PSO), relax both
    - $W \rightarrow R$
    - $W \rightarrow W$

Example 3

P1

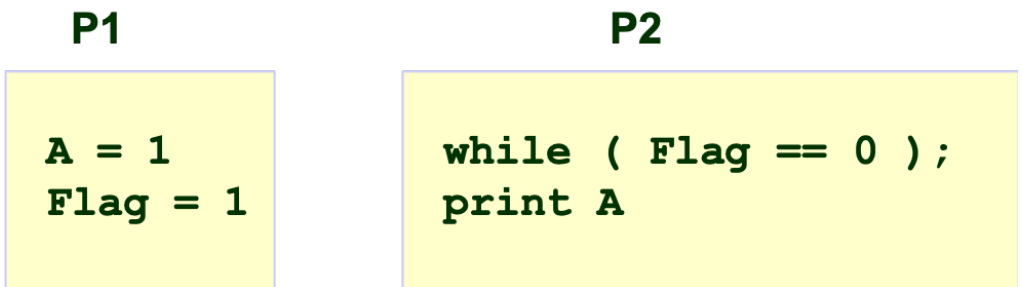
```
A = 1
Flag = 1
```

P2

```
while ( Flag == 0 );
print A
```

- A = Flag = 0 initially
- Can A = 0 be printed under the models of SC/TSO/PC/PSO?

Example 3 (cont'd)



- A = Flag = 0 initially
- Can A = 0 be printed under the models of SC/TSO/PC/PSO?
- *Possible for PSO: W(Flag), R(Flag), W(A), R(A)*

More For Your Own Exploration

- Other weak ordering models, e.g., causal consistency
  - No completion order of the memory operations is guaranteed, i.e., relax R → R, R → W
  - That is *out-of-order* execution
  - It is possible to decouple consistency model presented to programmer from that of the hardware/compiler: What is the memory consistency model of C++/Python?
- How to fully enjoy the performance benefits while guaranteeing program correctness?
  - Lock/Unlock
  - Memory fence
  - Volatile
  - ...

Homework Question

- If we declare every variable with volatile in Java...
- Which model do we essentially enforce: SC/TSO/PC/PSO?

For more on Java Memory Model, checkout "17.4 Memory Model" at ▸ Link