



# CS 152/252A Computer Architecture and Engineering



Sophia Shao

## Lecture 22: Memory Consistency Models

Google's Cloud TPU v4 provides exaFLOPS-scale ML

Happily for architects, machine learning (ML) models continue to evolve in challenging ways, both in scale and algorithmically (see Table 1 and Section 7.7). Examples of the former are large language models (LLMs) and examples of the latter are the embeddings necessary for recommender systems (Deep learning recommendation models or DLRMs) and the huge calculations of Transformers and BERT. The incredible scale of recent LLMs [Bro20,Tho22,Nay21] has stretched our ML supercomputer scale from 256 TPU v2 nodes to 4096 TPU v4 nodes.

training on TPU v4s over 30 days in October 2022.

<i>DNN Model</i>	<i>TPU v1</i> 7/2016 (Inference)	<i>TPU v3</i> 4/2019 (Training & Inference)	<i>TPU v4 Lite</i> 2/2020 (Inference)	<i>TPU v4</i> 10/2022 (Training)
MLP/DLRM	61%	27%	25%	24%
RNN	29%	21%	29%	2%
CNN	5%	24%	18%	12%
Transformer	--	21%	28%	57%
( <i>BERT</i> )	--	--	(28%)	(26%)
( <i>LLM</i> )	--	--	--	(31%)

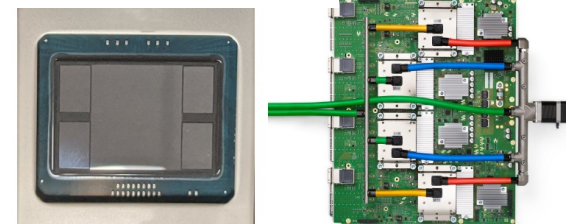


Figure 2: The TPU v4 package (ASIC in center plus 4 HBM stacks) and printed circuit board (PCB) with 4 liquid-cooled packages. The board's front panel has 4 top-side PCIe connectors and 16 bottom-side OSFP connectors for inter-tray ICI links.

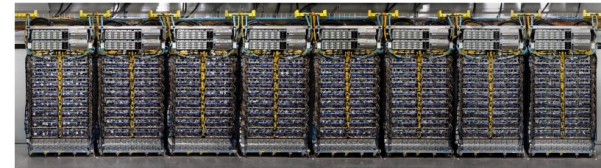


Figure 3: Eight of 64 racks for one 4096-chip supercomputer.



# Last Time in Lecture

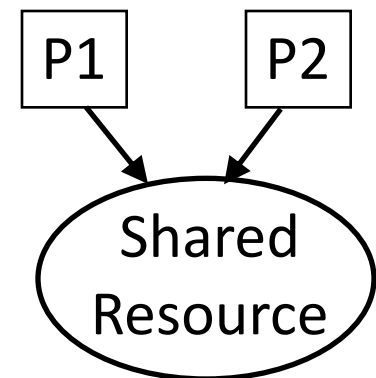
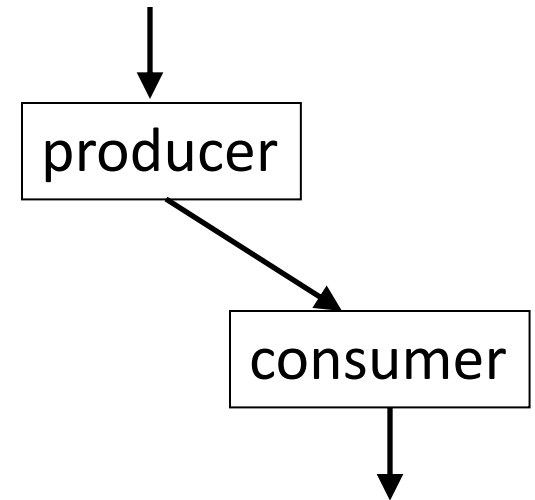
- Cache coherence, making sure every store to memory is eventually visible to any load to same memory address
- Cache line states: M,S,I or M,E,S,I
  - Cache miss if tag not present, or line has wrong state
- Snoopy coherence:
  - Broadcast updates and probe all cache tags on any miss of any processor, used to be bus connection now often broadcast over point-to-point links
  - Lower latency, but consumes lots of bandwidth on both the communication bus and for probing the cache tags
- Directory coherence:
  - Structure keeps track of which caches can have copies of data, and only send messages/probes to those caches
  - Complicated to get right with all the possible overlapping cache transactions

# Synchronization

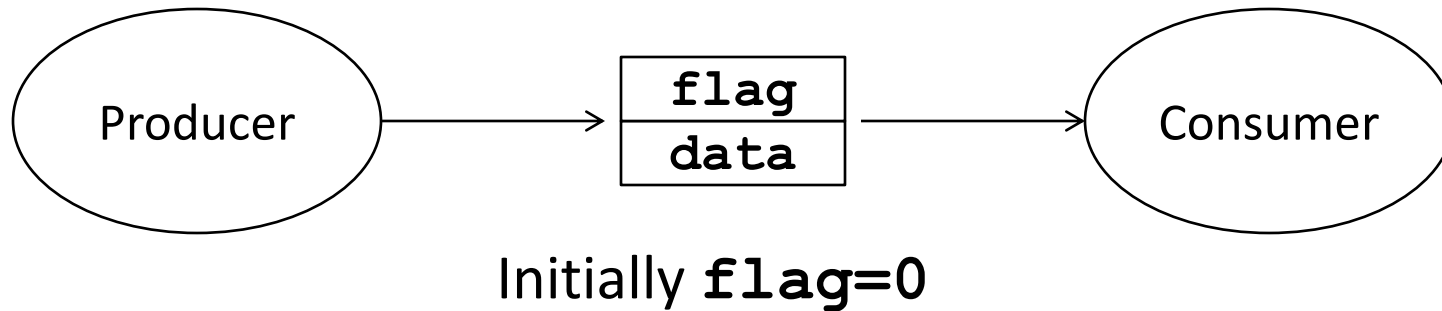
The need for synchronization arises whenever there are concurrent processes in a system (*even in a uniprocessor system*).

Two classes of synchronization:

- *Producer-Consumer*: A consumer process must wait until the producer process has produced data
- *Mutual Exclusion*: Ensure that only one process uses a resource at a given time



# Simple Producer-Consumer Example



```
sw xdata, (xdatap)
li xflag, 1
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
      beqz xflag, spin
      lw xdata, (xdatap)
```

Can consumer read **flag=1** before **data**  
written by producer visible to consumer?

# Memory Consistency Model

- Sequential ISA only specifies that each processor sees its own memory operations in program order
- Memory consistency model describes what values can be returned by load instructions across multiple hardware threads
- *Coherence* describes the legal values a *single* memory address should return
- *Consistency* describes properties across *all* memory addresses

# Coherence vs. Consistency

- The goal of cache coherence is to ensure that the memory system in a parallel computer behaves **as if the caches were not there**
  - A system w/o caches would have no need for cache coherence
- Memory consistency defines the allowed behavior of loads and stores **to different addresses** in a parallel system
  - The allowed behavior should be specified **whether or not caches are present**

# Multiprocessor Execution

Initially  $M[X] = M[Y] = 0$

P1:

(1)  $M[X] = 1$

(2)  $x2 = M[Y]$

P2:

(3)  $M[Y] = 1$

(4)  $x2 = M[X]$

## Possible Outcomes

P1.x2	P2.x2	
0	0	
0	1	
1	0	
1	1	

# Multiprocessor Execution

Initially  $M[X] = M[Y] = 0$

P1:

(1)  $M[X] = 1$

(2)  $x2 = M[Y]$

P2:

(3)  $M[Y] = 1$

(4)  $x2 = M[X]$

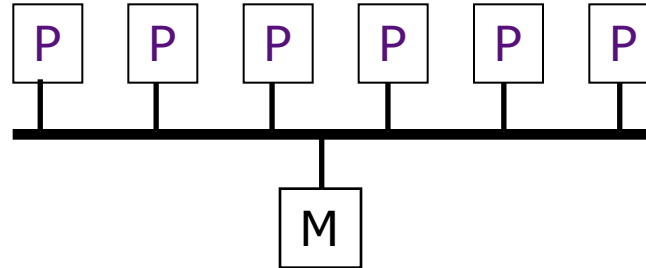
## Possible Outcomes

P1.x2	P2.x2	SC
0	0	N
0	1	Y
1	0	Y
1	1	Y



# Sequential Consistency (SC)

## *A Memory Model*



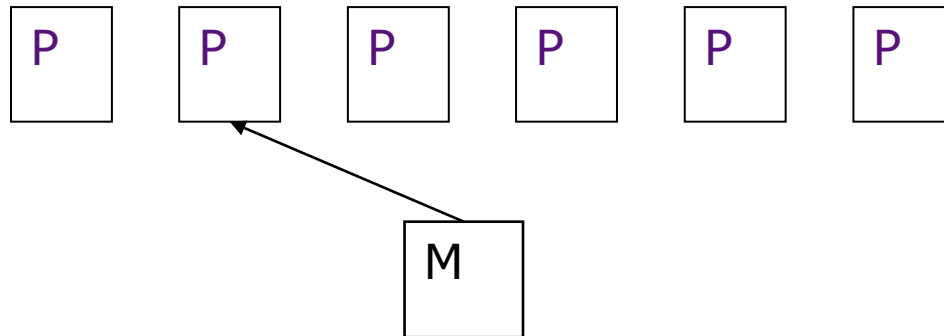
“ A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

*Leslie Lamport*

Sequential Consistency = arbitrary *order-preserving interleaving* of memory references of sequential programs

# Sequential consistency (switch metaphor)

- All processors issue loads and stores in program order
- Memory chooses a processor at random, performs a memory operation to completion, then chooses another processor, ...



Initially  $M[X] = M[Y] = 0$

P1:

(1)  $M[X] = 1$

(2)  $x2 = M[Y]$

P2:

(3)  $M[Y] = 1$

(4)  $x2 = M[X]$

P1.x2	P2.x2	SC
0	0	N
0	1	Y
1	0	Y
1	1	Y

# Simple Producer-Consumer Example



Initially flag = 0

**sw xdata, (xdatap)**  
**li xflag, 1**  
**sw xflag, (xflagp)**

**spin: lw xflag, (xflagp)**  
**beqz xflag, spin**  
**lw xdata, (xdatap)**

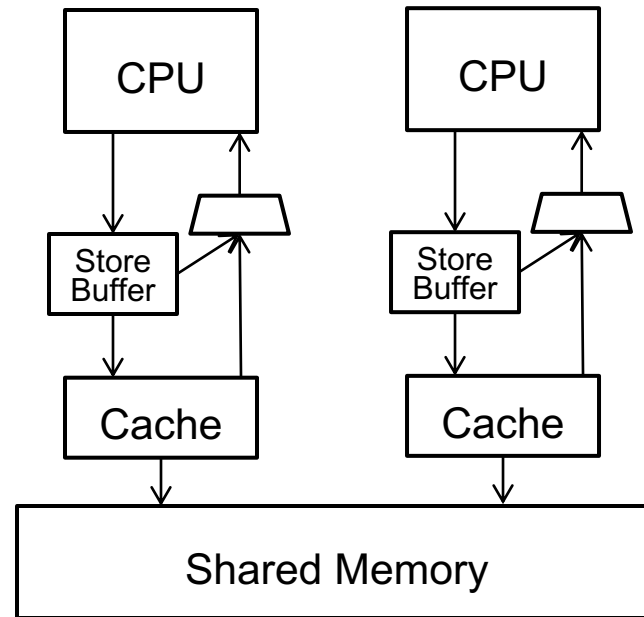
Dependencies from sequential ISA

Dependencies added by sequentially consistent memory model

# Most real machines are not SC

- Only a few commercial ISAs require SC
  - Neither IBM 370 nor x86 nor ARM nor RISC-V are SC
- Originally, architects developed uniprocessors with optimized memory systems (e.g., store buffer)
- When uniprocessors were lashed together to make multiprocessors, resulting machines were not SC
- Requiring SC would make simpler machines slower, or requires adding complex hardware to retain performance
- Architects/language designers/applications developers work hard to explain weak memory behavior
- Resulted in “weak” memory models with fewer guarantees

# Store Buffer



- Common optimization allows stores to be buffered while waiting for access to shared memory
- Load optimizations:
  - Later loads can go ahead of buffered stores if to different address
  - Later loads can bypass value from earlier buffered store if to same address

# TSO example

- Allows local buffering of stores by processor

Initially  $M[X] = M[Y] = 0$

P1:

(1)  $M[X] = 1$

(2)  $x2 = M[Y]$

P2:

(3)  $M[Y] = 1$

(4)  $x2 = M[X]$

## Possible Outcomes

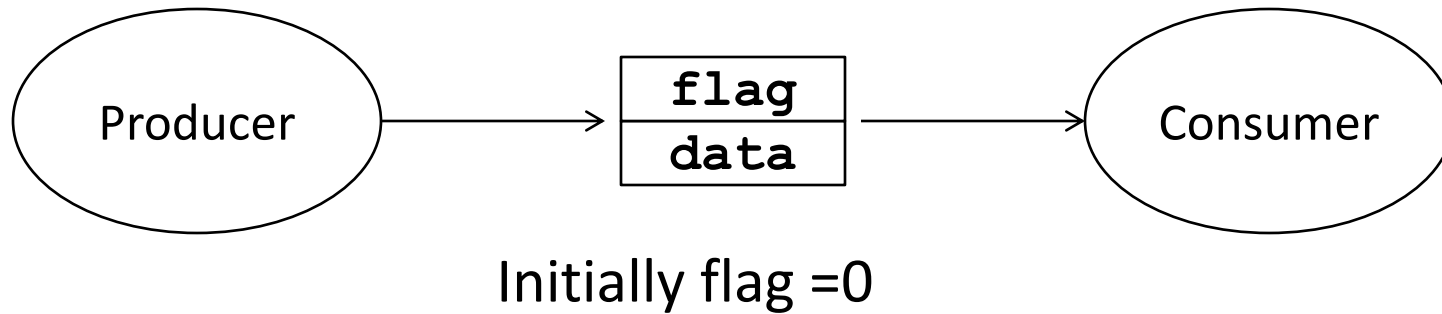
P1.x2	P2.x2	SC	TSO
0	0	N	Y
0	1	Y	Y
1	0	Y	Y
1	1	Y	Y

- TSO is the strongest memory model in common use

# Strong versus Weak Memory Consistency Models

- Stronger models provide more guarantees on ordering of loads and stores across different hardware threads
  - Easier ISA-level programming model
  - Can require more hardware to ensure orderings (e.g., MIPS R10K was SC, with hardware to speculate on load/stores and squash when ordering violations detected across cores)
- Weaker models provide fewer guarantees
  - Much more complex ISA-level programming model
    - Extremely difficult to understand, even for experts
  - Simpler to achieve high performance, as weaker models allow many hardware reorderings to be exposed to software
  - Additional instructions (fences) are provided to allow software to specify which orderings are required

# Fences in Producer-Consumer Example



```
sw xdata, (xdatap)
```

```
li xflag, 1
```

```
fence w,w //Write-write fence
```

```
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
```

```
beqz xflag, spin
```

```
fence r,r // Read-read fence
```

```
lw xdata, (xdatap)
```



# CS152 Administrivia

- Lab 4 due 4/18
- HW 5 due 4/27
- Midterm 2 Tuesday 4/11
  - covers lectures 11-18, plus associated problem sets, labs, and readings
  - No lecture next Tuesday
  - Discussions this week will go over midterm 2 review

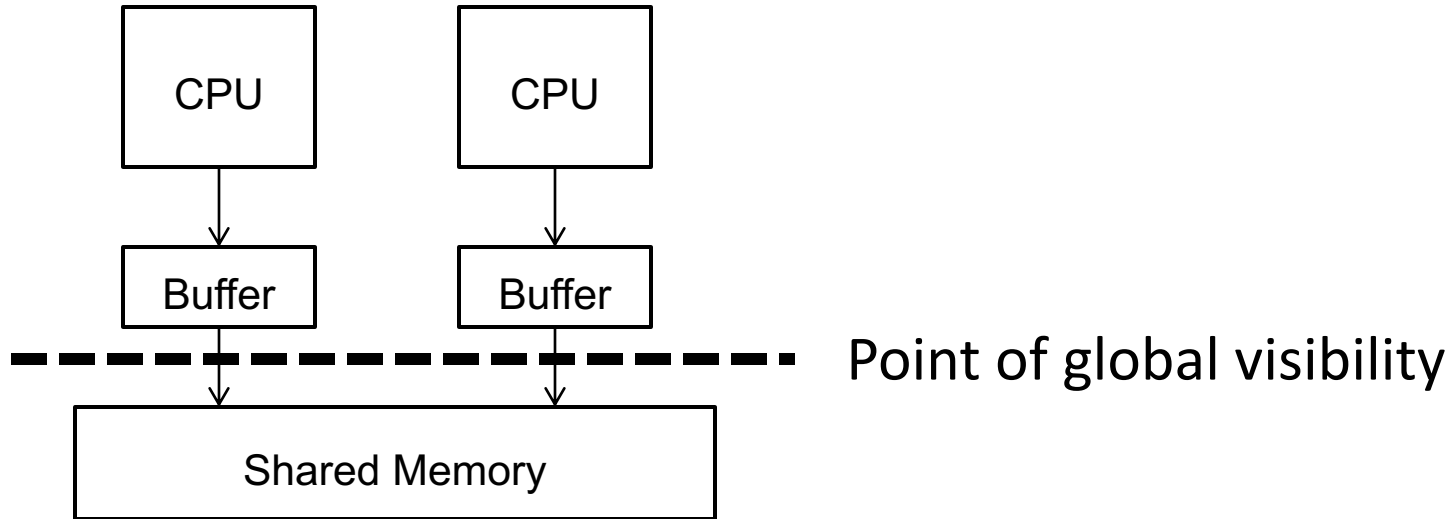
# CS252 Administrivia

- No reading next week
  - Good luck with midterm!

# Range of Memory Consistency Models

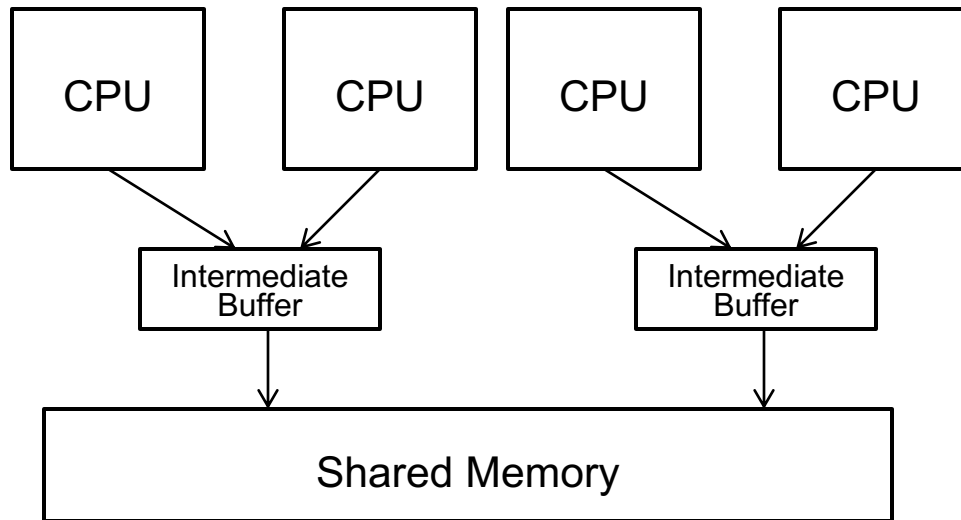
- SC “Sequential Consistency”
  - MIPS R10K
- TSO “Total Store Order”
  - *processor can see its own writes before others do (store buffer)*
  - IBM-370 TSO, x86 TSO, SPARC TSO (default), RISC-V RVTSO (optional)
- Weak, multi-copy-atomic memory models
  - *all processors see writes by another processor in same order*
  - Revised ARM v8 memory model
  - RISC-V RVWMO, baseline weak memory model for RISC-V
- Weak, non-multi-copy-atomic memory models
  - *processors can see another’s writes in different orders*
  - ARM v7, original ARM v8
  - IBM POWER
  - Digital Alpha (extremely weak MCM)
  - Recent consensus is that these appear to be too weak for general-purpose processors

# Multi-Copy Atomic models



- Each hardware thread must view its own memory operations in program order, but can buffer these locally and reorder accesses around the buffer
- But once a local store is made visible to one other hardware thread in system, all other hardware threads must also be able to observe it (this is what is meant by “atomic”)

# Hierarchical Shared Buffering



- Common in large systems to have shared intermediate buffers on path between CPUs and global memory
- Potential optimization is to allow some CPUs see some writes by a CPU before other CPUs
- Shared memory stores are not seen to happen atomically by other threads (non multi-copy atomic)

# Non-Multi-Copy Atomic

Initially  $M[X] = M[Y] = 0$

P1:	P2:	P3:
li x1, 1	lw x1, X	lw x1, Y
sw x1, X	sw x1, Y	fence r,r
		lw x2, X

Can  $P3.x1 = 1$ , and  $P3.x2 = 0$  ?

- In general, Non-MCA is very difficult to reason about
- Software in one thread cannot assume all data it sees is visible to other threads, so how to share data structures?
- Adding local fences to require ordering of each thread's accesses is insufficient – need a more global memory barrier to ensure all writes are made visible

# Relaxed Memory Models

- Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies where needed
- Which dependencies are dropped depends on the particular memory model
  - IBM370, TSO, PSO, WO, PC, Alpha, RMO, ...
  - Some ISAs allow several memory models, some machines have switchable memory models
- How to introduce needed dependencies varies by system
  - Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
  - Implicit effects of atomic memory instructions

*How on earth are programmers supposed to work with this????*

## But compilers reorder too!

```
//Producer code  
*datap = x/y;  
*flagp = 1;
```

```
//Consumer code  
while (!*flagp)  
    ;  
d = *datap;
```

- Compiler can reorder/remove memory operations:
  - Instruction scheduling, move loads before stores if to different address
  - Register allocation, cache load value in register, don't check memory
- Prohibiting these optimizations would result in very poor performance



# Language-Level Memory Models

- Programming languages have memory models too
- Hide details of each ISA's memory model underneath language standard
  - c.f. C function declarations versus ISA-specific subroutine linkage convention
- Language memory models: C/C++, Java
- Describe legal behaviors of threaded code in each language and what optimizations are legal for compiler to make
- E.g., C11/C++11: `atomic_load(memory_order_seq_cst)` maps to RISC-V `fence rw,rw; lw; fence r,rw`

# Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)