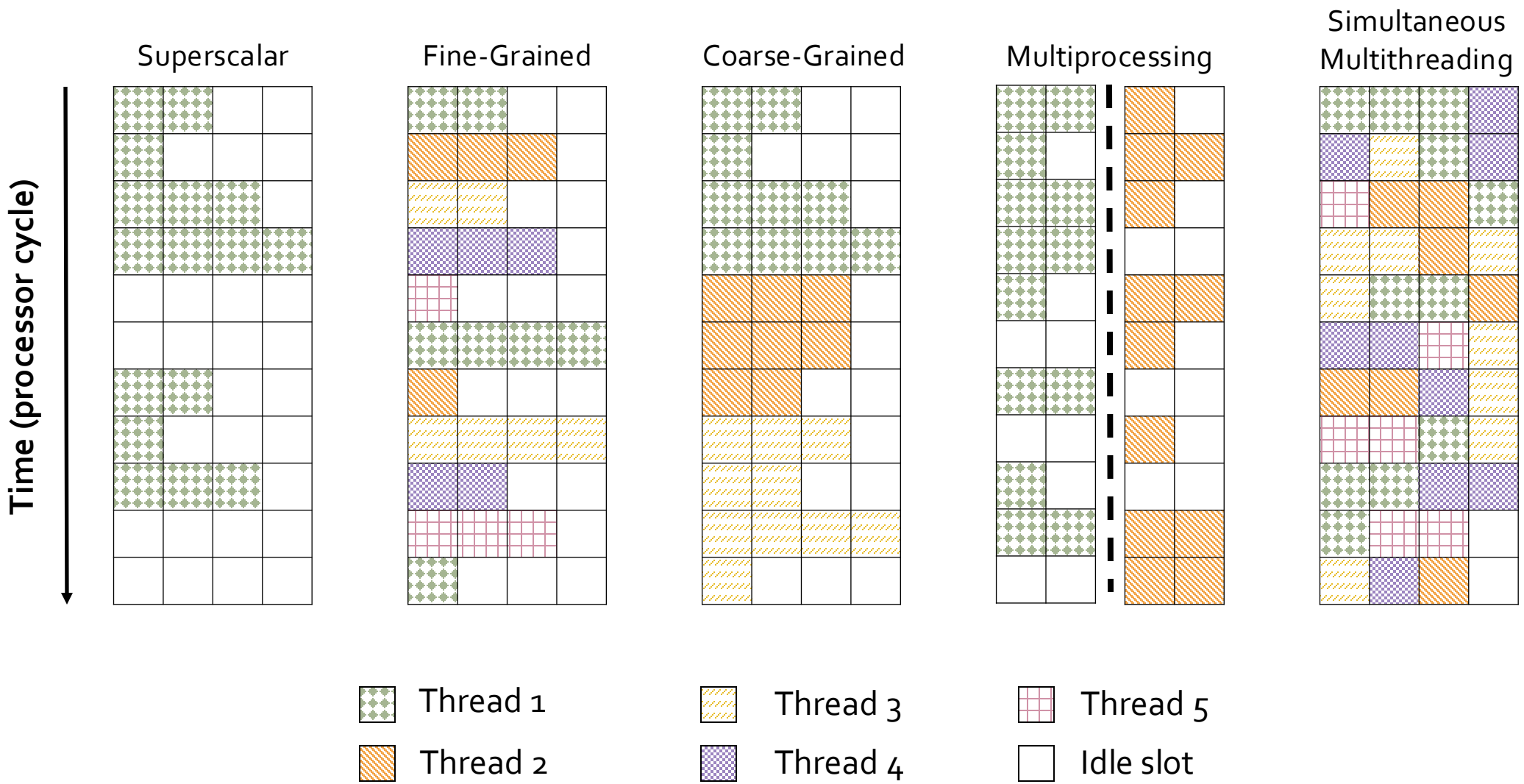
Computer Architecture

Memory Consistency and Synchronization

Ting-Jung Chang

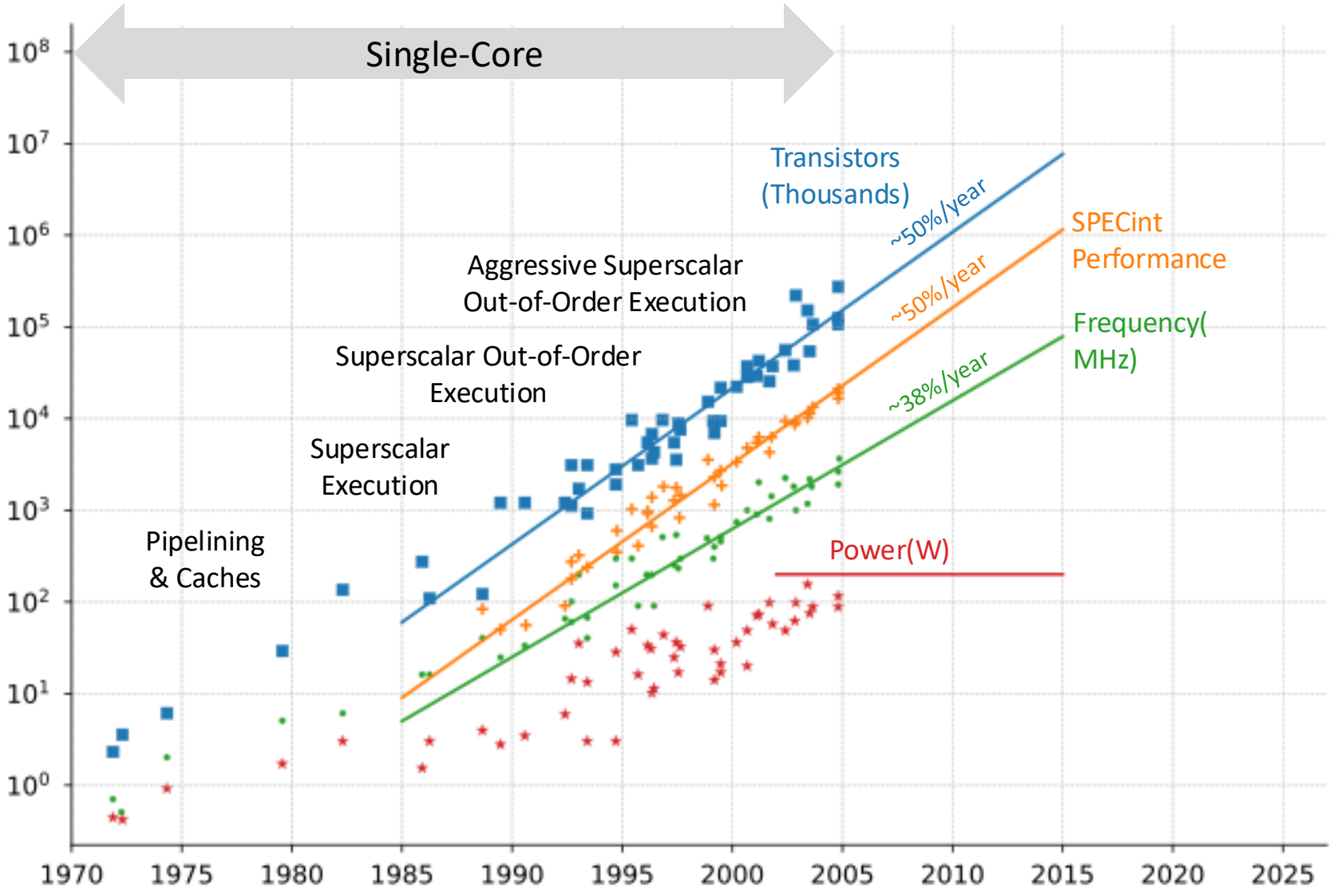
NYCU CS

Recap: Multithreaded Categories

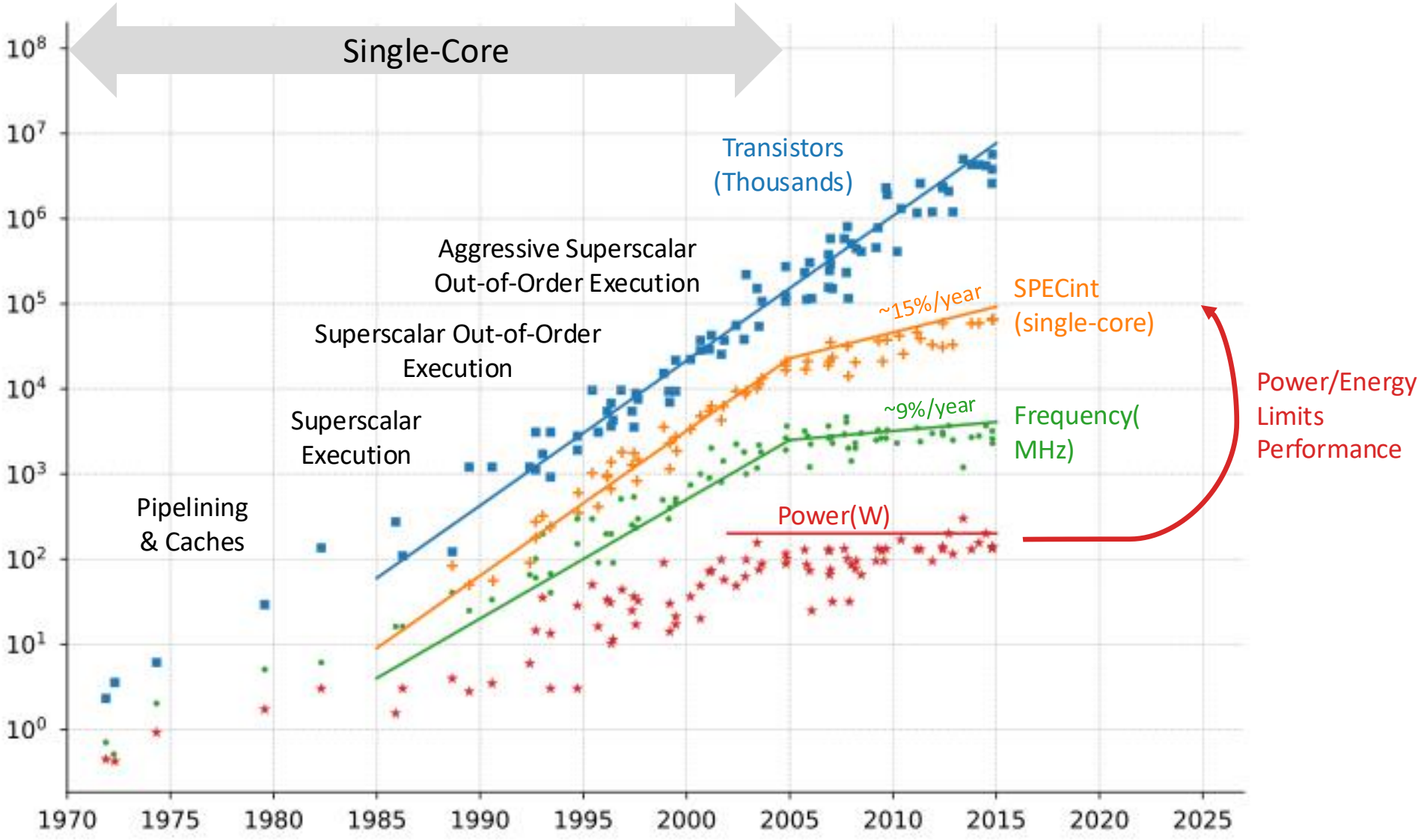


Recap: Meltdown and Spectre

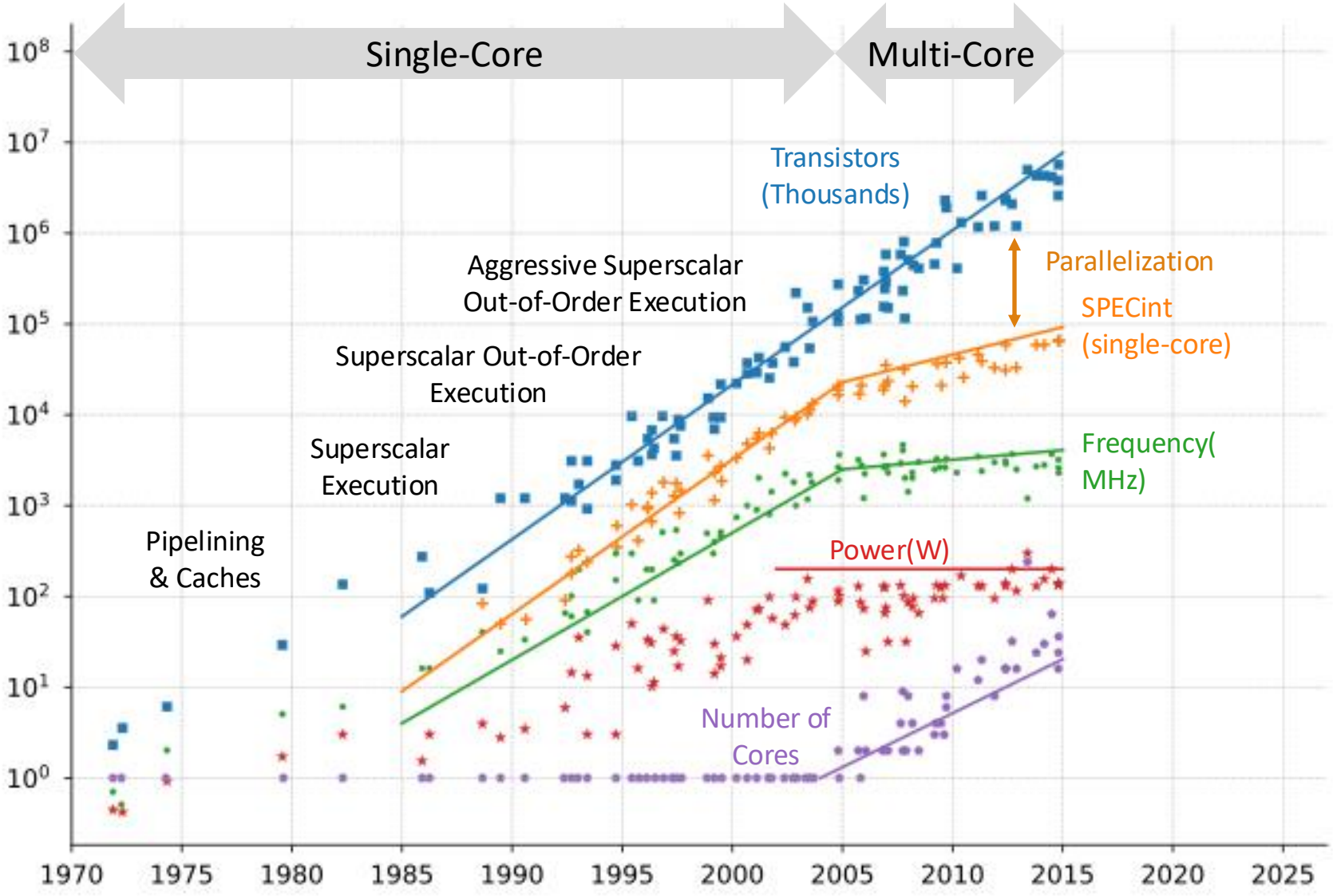
```
tingjung@arclab:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         52 bits physical, 57 bits virtual
  Byte Order:            Little Endian
CPU(s):                  48
Vulnerabilities:
  Gather data sampling:   Not affected
  Itlb multihit:          Not affected
  L1tf:                   Not affected
  Mds:                    Not affected
  Meltdown:               Not affected
  Mmio stale data:        Not affected
  Reg file data sampling: Not affected
  Retbleed:               Not affected
  Spec rstack overflow:   Mitigation; Safe RET
  Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prctl
  Spectre v1:             Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:             Mitigation; Enhanced / Automatic IBRS; IBPB conditional; STIBP always-on; RSB filling; PBRSB-eIBRS Not affected; BHI Not affected
  Srbds:                  Not affected
  Tsx async abort:        Not affected
```



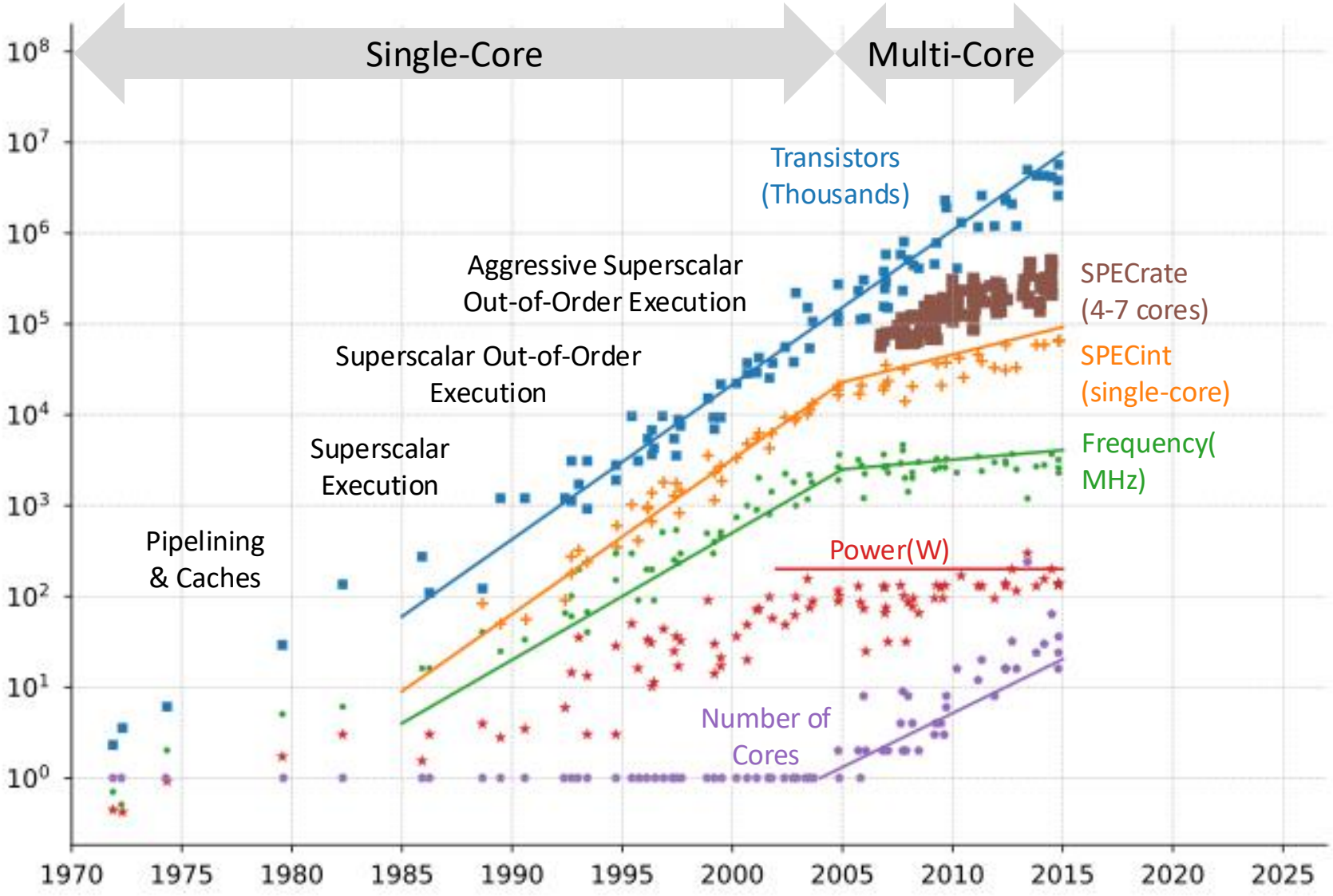
C. Batten, M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, K. Rupp & [Y. Shao, IEEE Micro'15] & [C. Leiserson, Science'20]



C. Batten, M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, K. Rupp & [Y. Shao, IEEE Micro'15] & [C. Leiserson, Science'20]

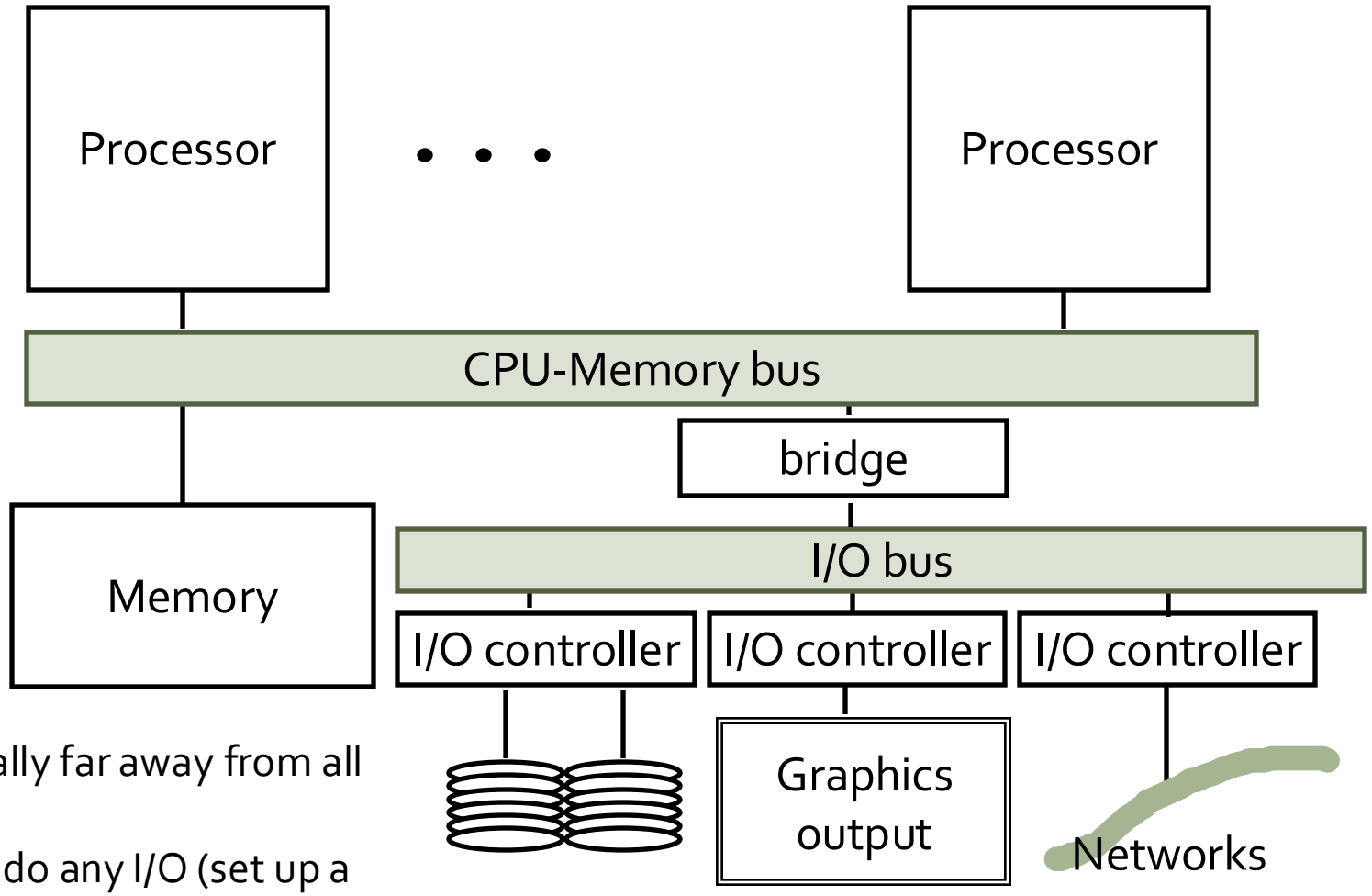


C. Batten, M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, K. Rupp & [Y. Shao, IEEE Micro'15] & [C. Leiserson, Science'20]



C. Batten, M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, K. Rupp & [Y. Shao, IEEE Micro'15] & [C. Leiserson, Science'20]

Symmetric Multiprocessors



- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

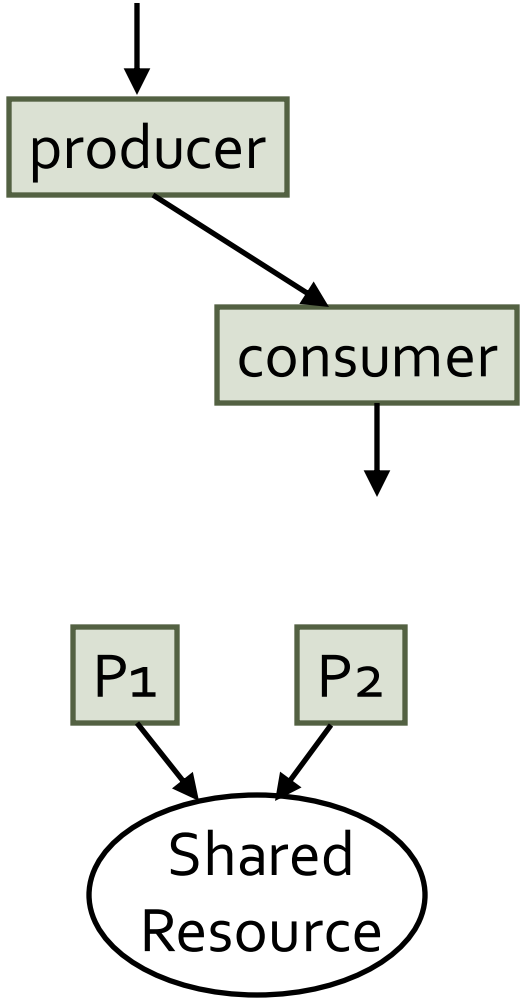
Synchronization

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

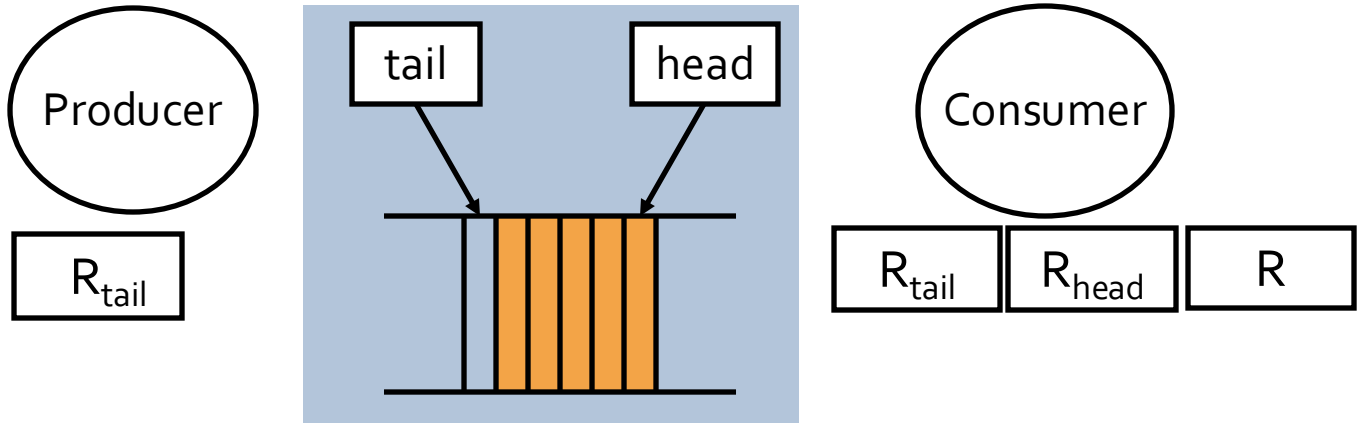
Two classes of synchronization:

- **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



A Producer-Consumer Example



Problems*?*

Producer posting Item x:

```
Load R_tail, (tail)
Store x, (R_tail)
R_tail=R_tail+1
Store R_tail, (tail)
```

Consumer

```
     Load R_head, (head)
spin: Load R_tail, (tail)
     if R_head==R_tail goto spin
     Load R, (R_head)
     R_head=R_head+1
     Store R_head, (head)
     process(R)
```

The program is written assuming instructions are executed in order.

Agenda

- Memory Consistency Models
- Synchronization

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

A Producer-Consumer Example

continued

Producer posting Item x:

```
    Load R_tail, (tail)
1   Store x, (R_tail)
    R_tail=R_tail+1
2   Store R_tail, (tail)
```

Consumer

```
    Load R_head, (head)
spin: Load R_tail, (tail)      3
    if R_head==R_tail goto spin
    Load R, (R_head)           4
    R_head=R_head+1
    Store R_head, (head)
    process(R)
```

Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

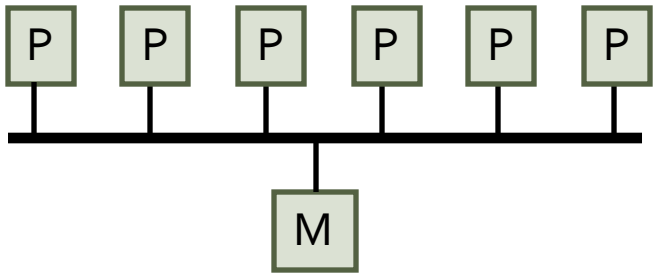
Problem sequences are:

2, 3, 4, 1

4, 1, 2, 3

Sequential Consistency

A Straightforward 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

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

```
T1:                                  T2:
    Store 1,  (X) (X =  1)               Load  R1, (Y)
    Store 11, (Y) (Y = 11)               Store R1, (Y') (Y'= Y)
                                         Load  R2, (X)
                                         Store R2, (X') (X'= X)
```

what are the legitimate answers for X' and Y' ?

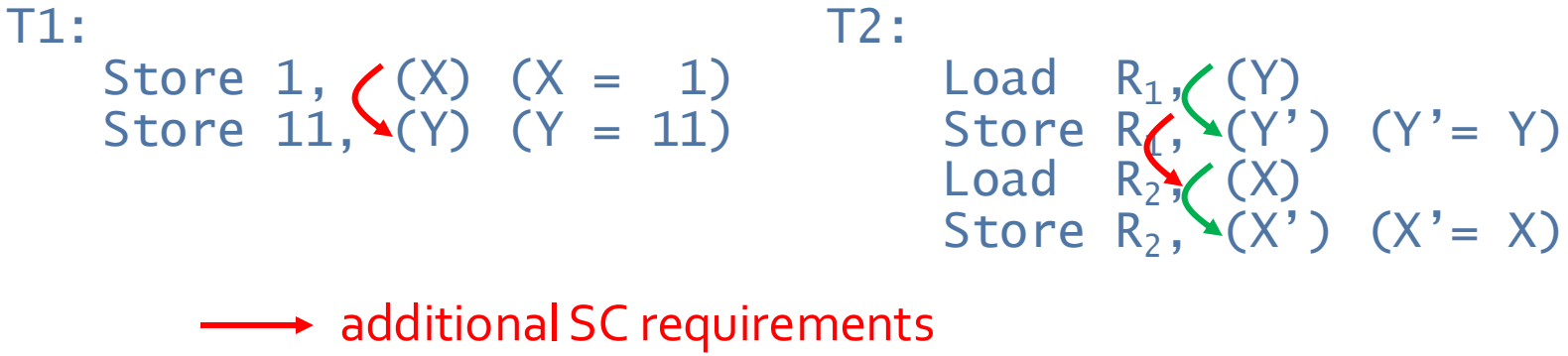
X'	Y'	SC
1	11	Y
0	10	Y
1	10	Y
0	11	N

If Y is 11 then X cannot be 0

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (→)

What are these in our example ?



→ additional SC requirements

Does (can) a system with caches or out-of-order execution capability provide a **sequentially consistent** view of the memory ?

SC is easy to understand, but architects and compiler writers want to violate it for performance

Memory Model Issues

- Architectural optimizations that are correct for uniprocessors often violate sequential consistency and result in a new memory model for multiprocessors

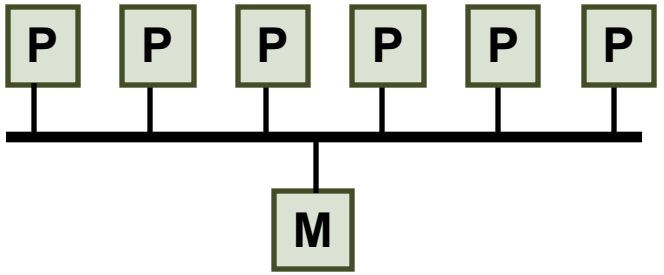
Most Real Machines are not SC

- 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

Strong vs. Weak MCMs

- 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

Issues in Implementing Sequential Consistency



- Implementation of SC is complicated by two issues
  - Out-of-order execution capability
    - Load(a); Load(b) yes
    - Load(a); Store(b) yes if a ≠ b
    - Store(a); Load(b) yes if a ≠ b
    - Store(a); Store(b) yes if a ≠ b
  - Caches
    - Caches can prevent the effect of a store from being seen by other processors

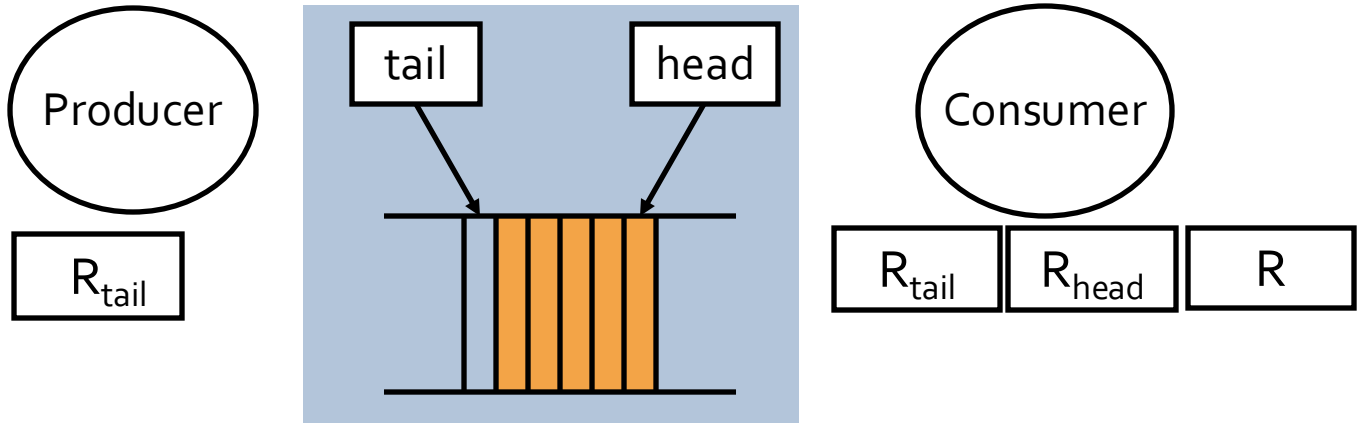
SC complications motivate architects to consider weak or relaxed memory models

Memory Fences

Instructions to sequentialize memory accesses

- Processors with **relaxed or weak memory models** permit Loads and Stores to different addresses to be reordered, remove some/all extra dependencies imposed by SC
  - LL, LS, SL, SS
- Need to provide **memory fence** instructions to force the serialization of memory accesses
- Examples of relaxed memory models:
  - **Total Store Order**: LL, LS, SS, enforce SL with fence
  - **Partial Store Order**: LL, LS, enforce SL, SS with fences
  - **Weak Ordering**: enforce LL, LS, SL, SS with fences
- Memory fences are expensive operations – mem instructions wait for all relevant instructions in-flight to complete (including stores to retire – need store acks)
- **However, cost of serialization only when it is required!**

Using Memory Fences



Producer posting Item x:

Load R_{tail}, (tail)

Store x, (R_{tail})

$MFence_{SS}$

$R_{tail}=R_{tail}+1$

Store R_{tail}, (tail)

ensures that tail ptr is not updated before x has been stored

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail)

if $R_{head}==R_{tail}$ goto spin

$Mfence_{LL}$

Load R, (R_{head})

$R_{head}=R_{head}+1$

Store R_{head}, (head)

process(R)

ensures that R is not loaded before x has been stored

Language-Level Memory Models

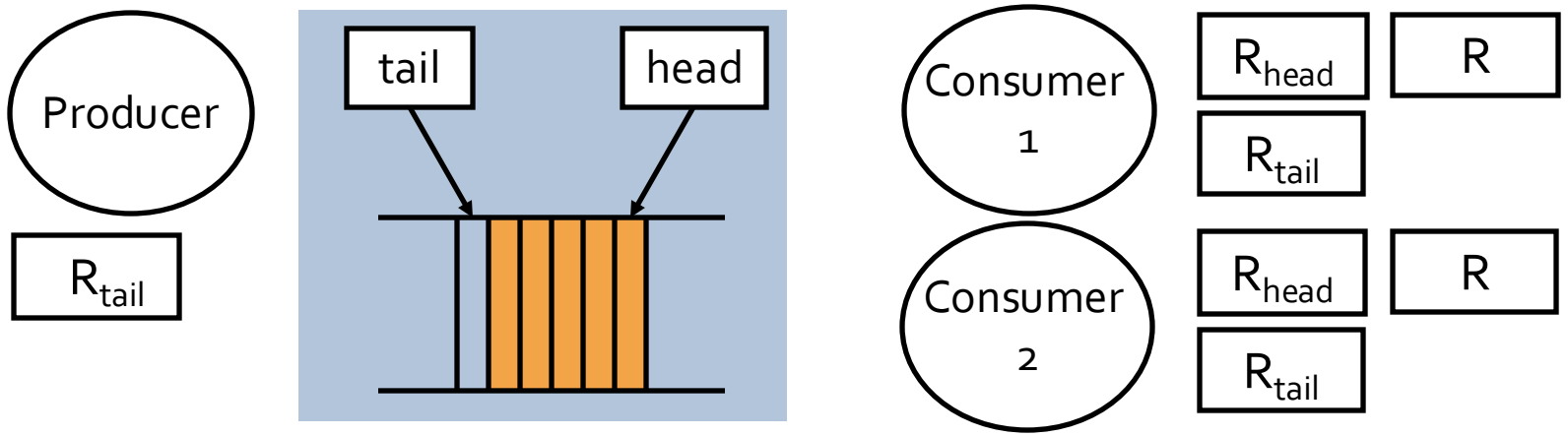
- Programming languages have memory models too
- Hide details of each ISA's memory model underneath language standard
  - c.f. C function declarations versus ISA-specific subroutine linkage convention

- 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`**

Agenda

- Memory Consistency Models
- Synchronization

Multiple Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)

Store x, (R_{tail})

R_{tail}=R_{tail}+1

Store R_{tail}, (tail)

Critical section:
Needs to be executed atomically by one consumer ⇒ locks

Consumer

spin: Load R_{head}, (head)

Load R_{tail}, (tail)

if R_{head}==R_{tail} goto spin

Load R, (R_{head})

R_{head}=R_{head}+1

Store R_{head}, (head)

process(R)

What is wrong with this code?
Assume SC

Locks or Semaphores

E. W. Dijkstra, 1965

- A semaphore is a non-negative integer, with the following operations:
  - P(s): if s>0, decrement s by 1, otherwise wait
    probeer te verlagen, literally ("try to reduce")
  - V(s): increment s by 1 and wake up one of the waiting processes
    verhogen ("increase")
- P's and V's must be executed atomically, i.e., without
  - interruptions or
  - interleaved accesses to s by other processors

```
Process i
        P(s)
           <critical section>
        V(s)
```

initial value of s determines
the maximum no. of processes
in the critical section

ISA Support for Semaphores

- Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design…

- Simpler solution:
  - atomic read-modify-write (RMW) instructions
  - atomic: **as one uninterruptible unit**
- Examples: m is a memory location, R is a register

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
        M[m] ← 1;
```

```
Fetch&Add (m), R_V, R:
    R ← M[m];
    M[m] ← R + R_V;
```

```
Swap (m), R:
    R_t ← M[m];
    M[m] ← R;
    R ← R_t;
```

Multiple Consumers Example

using the Test&Set Instruction

```
P:      Test&Set (mutex), R_temp
        if (R_temp!=0) goto P
        Load R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head=R_head+1
        Store R_head, (head)
V:      Store 0, (mutex)
        process(R)
```

Acquire Lock

Critical Section

Release Lock

Other atomic read-modify-write instructions (Swap, Fetch & Add, etc.) can also implement P's and V's

What if the process stops or is swapped out while in the critical section?

Executing Critical Sections without Locks

- If a software thread is descheduled after taking lock, other threads cannot make progress inside critical section
- "Non-blocking" synchronization allows critical sections to execute atomically without taking a lock

Nonblocking Synchronization

```
Compare&Swap(m), Rt, Rs:
 if (Rt==M[m])
     then M[m]=Rs;
          Rs=Rt ;
          status ← success;
     else status ← fail;
```

status is an implicit argument

```
try:    Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rnewhead=Rhead+1
        Compare&Swap(head), Rhead, Rnewhead
        if (status==fail) goto try
        process(R)
```

Compare-and-Swap Issues

- Compare and Swap is a complex instruction
  - Three source operands: address, comparand, new value
  - One return value: success/fail or old value
- ABA problem
  - Load(A), Y=process(A), success=CAS(A,Y)
  - What if different task switched A to B then back to A before process() finished?
- Solving ABA:
  - Add a counter, and make CAS access two words
- Double Compare and Swap (DCAS)
  - Five source operands: one address, two comparands, two values
  - Load(<A1,A2>), Z=process(A1), success=CAS(<A1,A2>,<Z s,A2+1>)

Load-reserve & Store-conditional

aka Load-link, Load-Locked

- Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs'
          reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-reserve R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head=R_head+1
        Store-conditional (head), R_head
        if (status==fail) goto try
        process(R)
```

LR/SC Issues

- LR/SC does not suffer from ABA problem, as any access to addresses will clear reservation regardless of value
  - CAS only checks stored values not intervening accesses
- LR/SC non-blocking synchronization can livelock between two competing processors
  - CAS guaranteed to make forward progress, as CAS only fails if some other thread succeeds
- RISC-V LR/SC makes guarantee of forward progress provided code inside LR/SC pair obeys certain rules
  - Can implement CAS inside RISC-V LR/SC

Performance of Locks

- Blocking atomic read-modify-write instructions
  - e.g., Test&Set, Fetch&Add, Swap
- Non-blocking atomic read-modify-write instructions
  - e.g., Compare&Swap, Load-reserve/Store-conditional
- Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:

- degree of contention,
- caches,
- out-of-order execution of Loads and Stores

Mutual Exclusion Using Load/Store

(assume SC)

A protocol based on two shared variables c1 and c2.
Initially, both c1 and c2 are 0 (not busy)

Process 1

```
   ...
   c1=1;
L: if c2==1 then go to L
      <critical section>
   c1=0;
```

Process 2

```
   ...
   c2=1;
L: if c1==1 then go to L
      <critical section>
   c2=0;
```

What is wrong?

Deadlock!

Mutual Exclusion

second attempt

To avoid deadlock, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

Process 1

```
   ...
L: c1=1;
   if c2==1 then
      {c1=0; go to L}
   <critical section>
   c1=0;
```

Process 2

```
   ...
L: c2=1;
   if c1==1 then
       {c2=0; go to L}
   <critical section>
   c2=0;
```

- Deadlock is not possible but with a low probability a **livelock** may occur.
- An unlucky process may never get to enter the critical section → **starvation**

A Protocol for Mutual Exclusion

T. Dekker, 1966 (Idea) G. Peterson (As Shown)

A protocol based on 3 shared variables c1, c2 and turn.
Initially, both c1 and c2 are 0 (not busy)

Process 1

```
   ...
   c1=1;
   turn=1;
L: if c2==1 && turn==1
      then go to L
   <critical section>
   c1=0;
```

Process 2

```
   ...
   c2=1;
   turn=2;
L: if c1==1 && turn==2
      then go to L
   <critical section>
   c2=0;
```

- turn == i ensures that only process i can wait
- variables c1 and c2 ensure mutual exclusion

Solution for n processes was given by Dijkstra and is quite tricky!

N-process Mutual Exclusion

Lamport's Bakery Algorithm

```
Process i                Initially num[j] = 0, for all j
```

Entry Code

```
    choosing[i] = 1;
    num[i] = max(num[0], …, num[N-1]) + 1;
    choosing[i] = 0;

    for(j = 0; j < N; j++)  {
        while( choosing[j] );
        while( num[j] &&
               (( num[j] < num[i] ) ||
                 ( num[j] == num[i] &&  j < i )));
    }
```

Exit Code

```
    num[i] = 0;
```

ISA Support for Mutual-Exclusion Locks

- Regular loads and stores in SC model (plus fences in weaker model) sufficient to implement mutual exclusion, but code is inefficient and complex
- Therefore, atomic read-modify-write (RMW) instructions added to ISAs to support mutual exclusion
- Many forms of atomic RMW instruction possible, some simple examples:
  - Test and set (reg_x = M[a]; M[a]=1)
  - Swap (reg_x=M[a]; M[a] = reg_y)

RISC-V Atomic Memory Operations

- Atomic Memory Operations (AMOs) have two ordering bits:
  - Acquire (aq)
  - Release (rl)
- If both clear, no additional ordering implied
- If aq set, then AMO "happens before" any following loads or stores
- If rl set, then AMO "happens after" any earlier loads or stores
- If both aq and rl set, then AMO happens in program order

Summary

- SC is too low level a programming model. High-level programming should be based on critical sections & locks, atomic transactions, monitors, …
- High-level parallel programming should be oblivious of memory model issues
  - Programmer should not be affected by changes in the memory model
- ISA definition for Load, Store, Memory Fence, synchronization instructions should
  - Be precise
  - Permit maximum flexibility in hardware implementation
  - Permit efficient implementation of high-level parallel constructs

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)
  - David Wentzlaff (Princeton)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- Cornell material derived from course ECE 4750
- Princeton material derived from course ECE 475