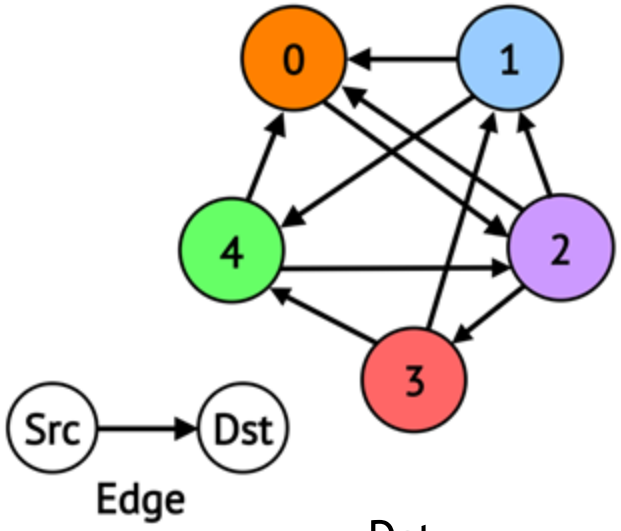
CMU 18-344: Computer Systems and the Hardware/Software Interface

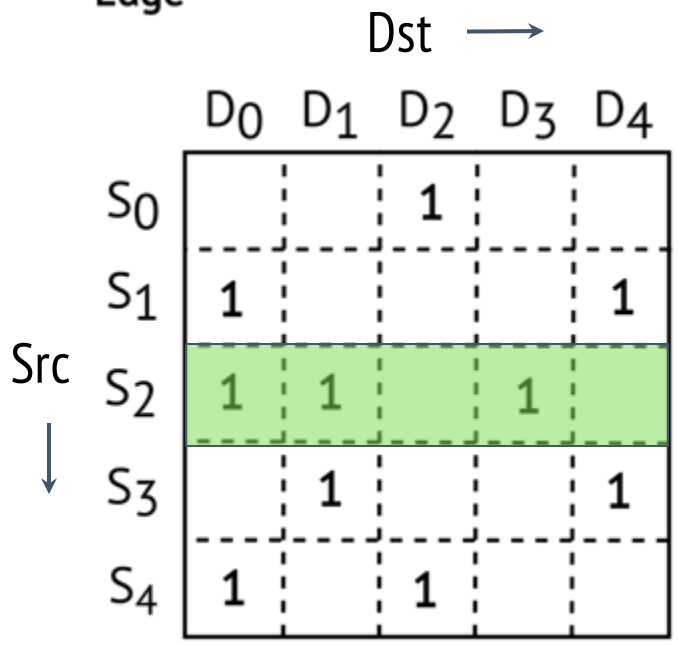
Fall 2024, Prof. Brandon Lucia

Recap: Sparse Problems

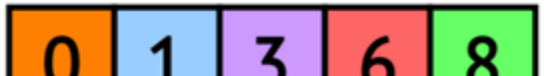
- What is a sparse problem? Why are they called “sparse”?
- What makes sparse problems hard?
- Roofline performance modeling
- Hardware and software strategies for optimizing sparse problems

Compressed Sparse Data Structures for Feasible Memory Size

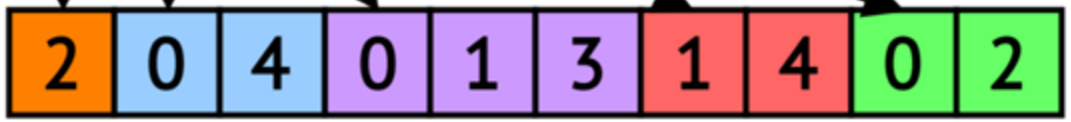




Offsets Array (OA)



Neighbors Array (NA)

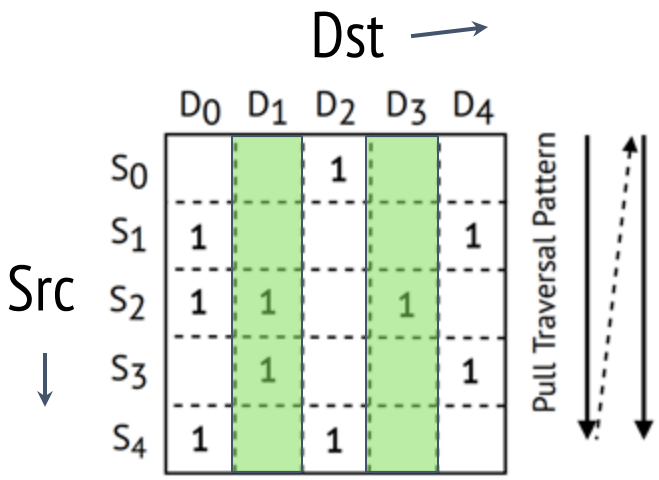


Compressed Sparse Row (CSR)
Outgoing Neighbors

Vertex Property Array
i.e., srcData / dstData

Often we will leave the vertex property array implicitly defined when we talk about sparse structures, but it is always there

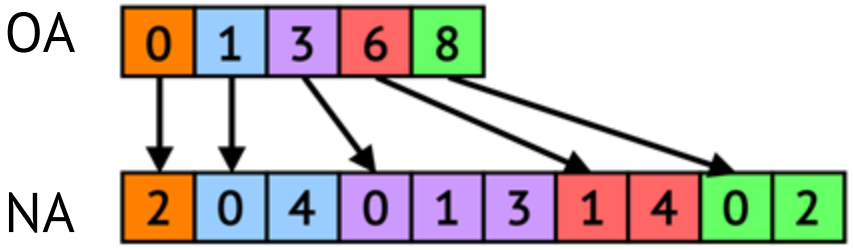
Compressed Representations ⇒ Irregular Memory Accesses



Push (CSR Traversal)

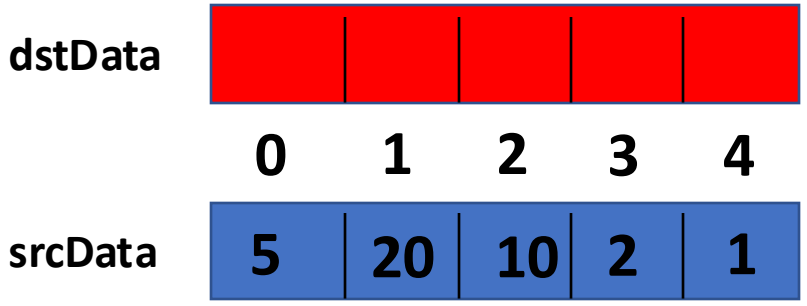
```
for src in G:
    for dst in out_neighs(src):
        dstData[dst] += srcData[src]
```

Push traversal performs *irregular write operations* that lack locality



CSR

i.e., x_{i+1}

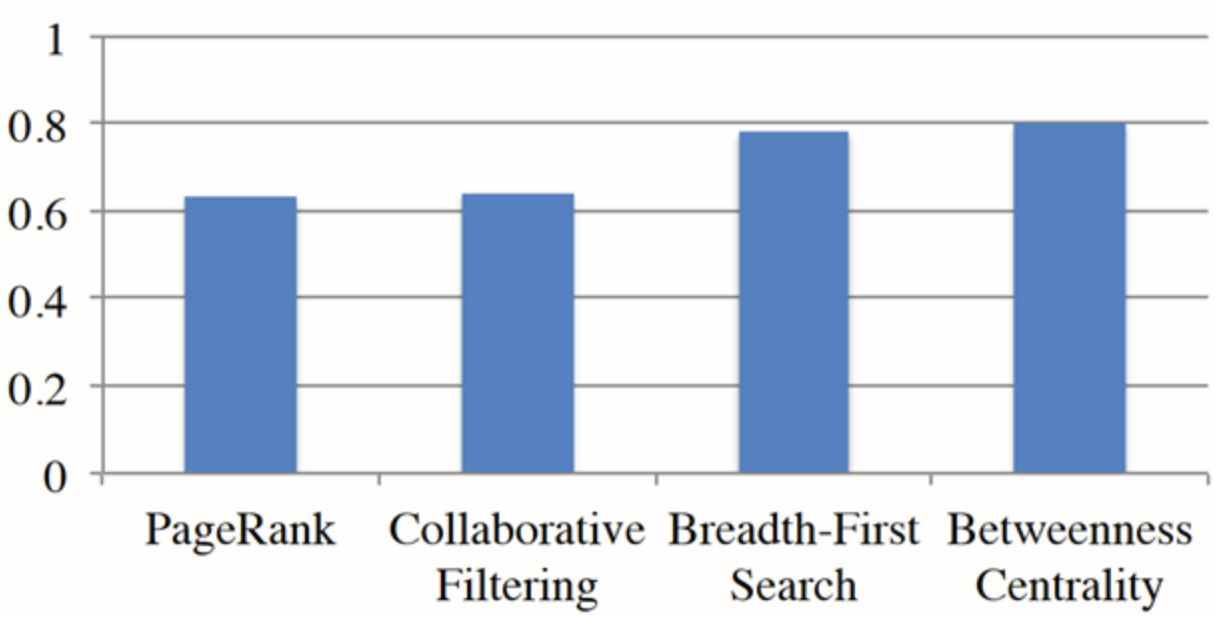


e.g., current rank of page I,
e.g., current shortest path
from source vertex

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

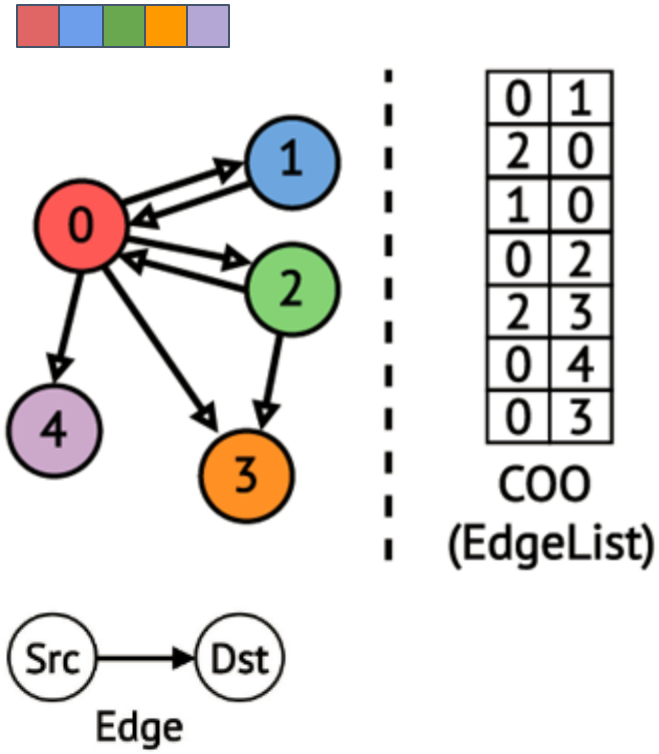
Cycles stalled on DRAM / Total Cycles



Problem: Sparse representations make processing large graphs feasible, but graph processing still entails a large working set with poor locality

Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Even Building the CSR / CSC is an Irregular Access Pattern!




```
for e in EL:
  neigh_count[e.dst]++; /*e.src*/
```

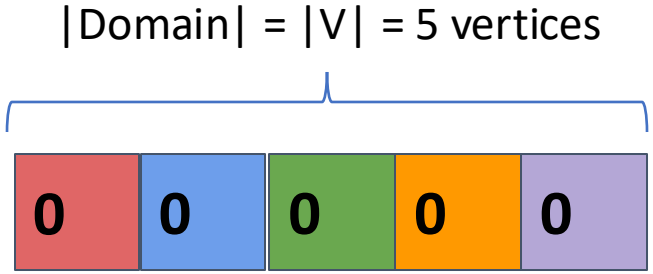
neigh_count

Updates to the neigh_count array are to random elements determined by order of edges in edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

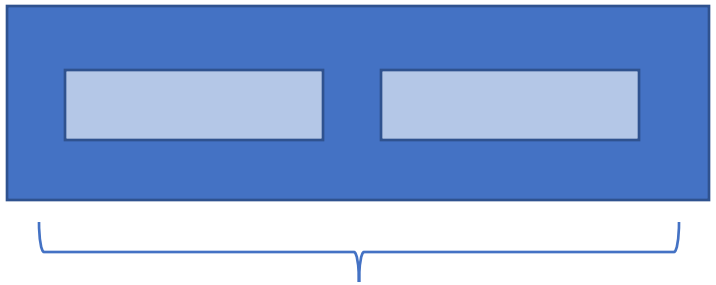
0	1
2	0
1	0
0	2
2	3
0	4
0	3

COO
(EdgeList)



Recall: irregular accesses into vertex data array based on e.dst *which are essentially random*

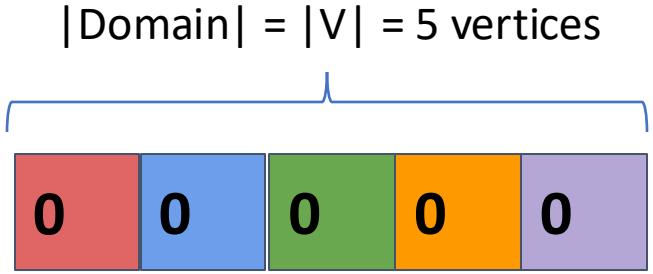
Bad for the cache: the size of the *domain* of vertex data array entries is |V|, but the cache holds only |C| << |V| entries



Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

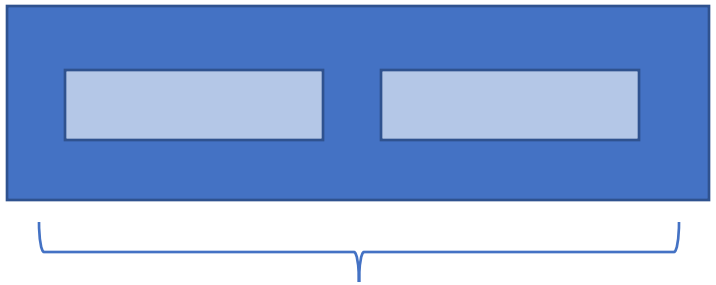
0	1
2	0
1	0
0	2
2	3
0	4
0	3

COO
(EdgeList)



Recall: irregular accesses into vertex data array based on e.dst *which are essentially random*

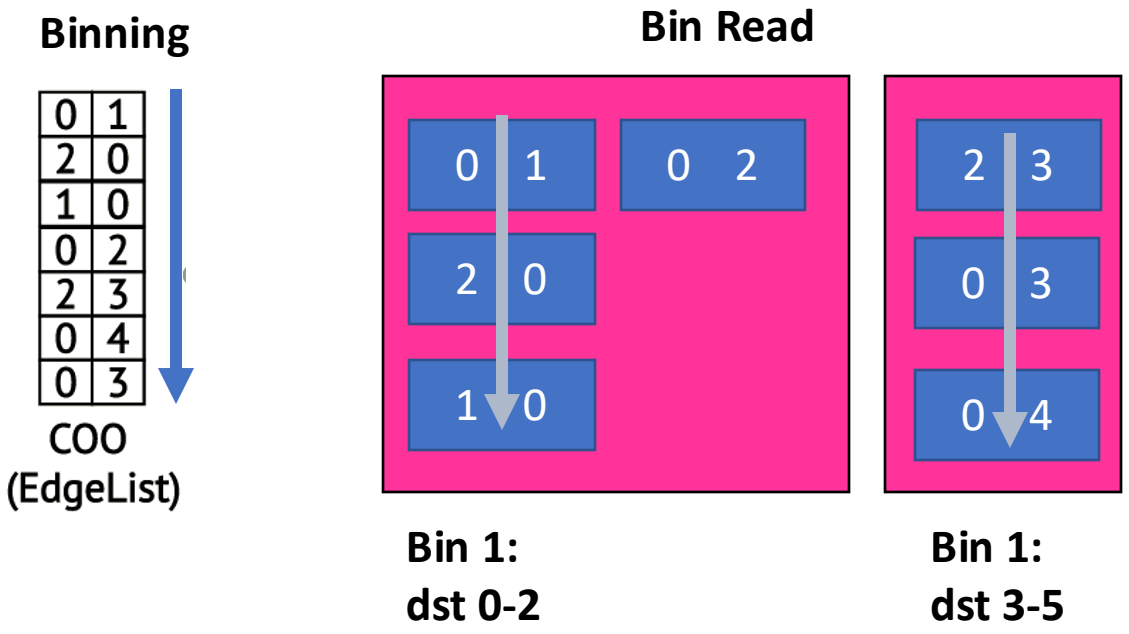
Bad for the cache: the size of the *domain* of vertex data array entries is |V|, but the cache holds only |C| << |V| entries



Key idea in propagation blocking: Limit the domain of updates to a *sub-space* of vertices, **V***, so that |V*| <= |C| and do multiple sub-spaces of V*s, so that all V*s together = V

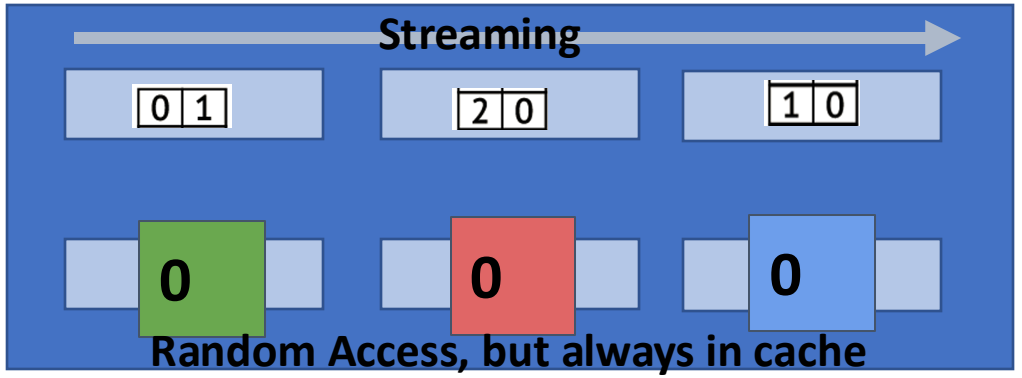
Propagation Blocking: Performance Analysis

Traverse the edge list twice instead of once



What about the performance of reading the edge list during binning?

Usually save a little space in cache for *streaming edge list* data. Easy to cache.



dstData

Remember: dstData[e.dst] ++

and e.dst is random, from edge list

Propagation Blocking

```
PropagationBlocking_EdgeCount(EdgeList E){

  Bins B[];
  for edge in E{
    add_to_bin( find_bin(edge) )
  }

  for bin in B{
    for e in bin{
      dstData[e.dst]++
    }
  }

}
```

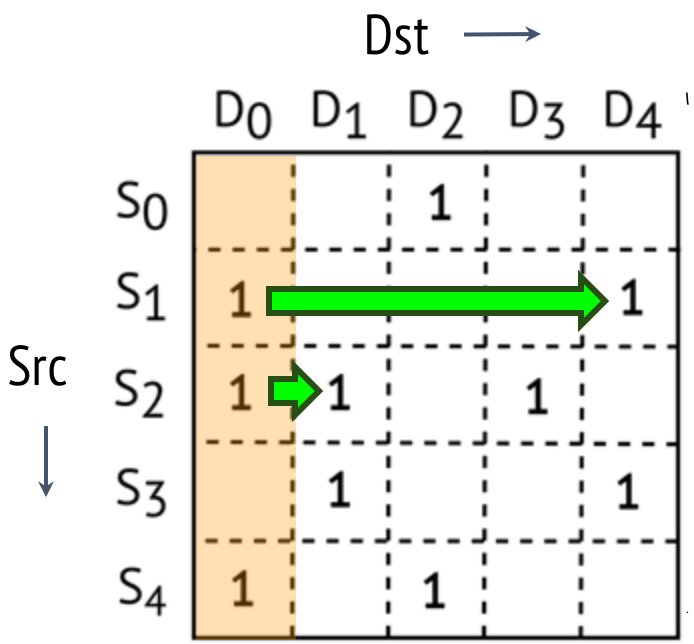
Reducing Pagerank Communication via Propagation Blocking

Scott Beamer*
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California
sbeamer@lbl.gov

Krste Asanović David Patterson
Electrical Engineering & Computer Sciences Department
University of California
Berkeley, California
{krste,pattrsn}@eecs.berkeley.edu

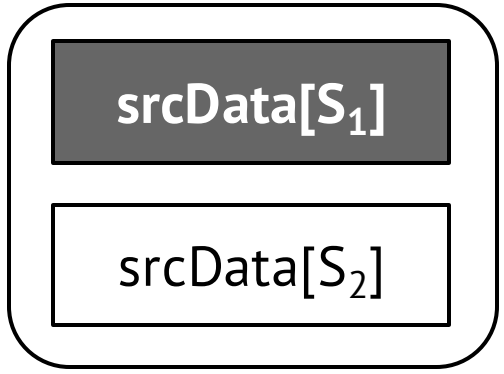
Application of Propagation Blocking for Graph Applications (Page Rank only, at first) discovered in 2017
(Prior work on “radix partitioning” applied the idea to other domains, but not graphs)

Using The Graph's Transpose For Optimal Replacement



Pull Execution (*CSC Traversal*)

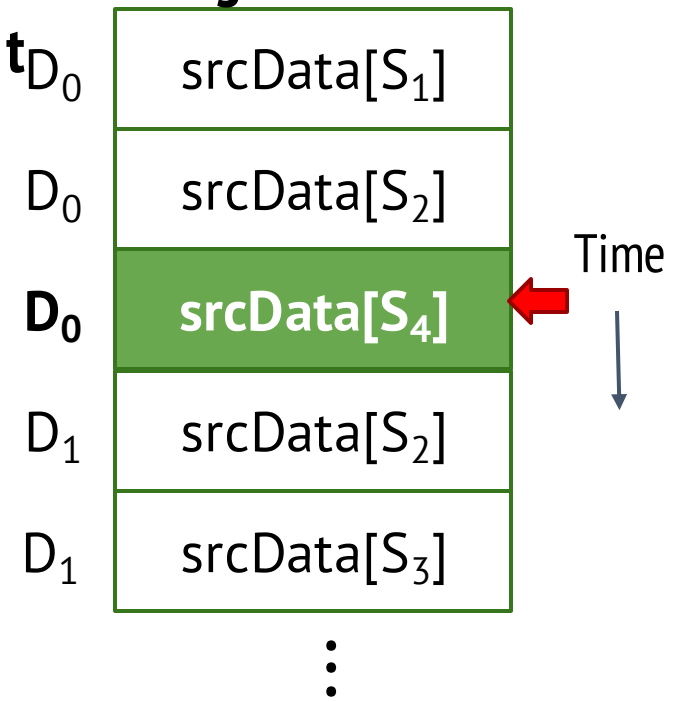
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```



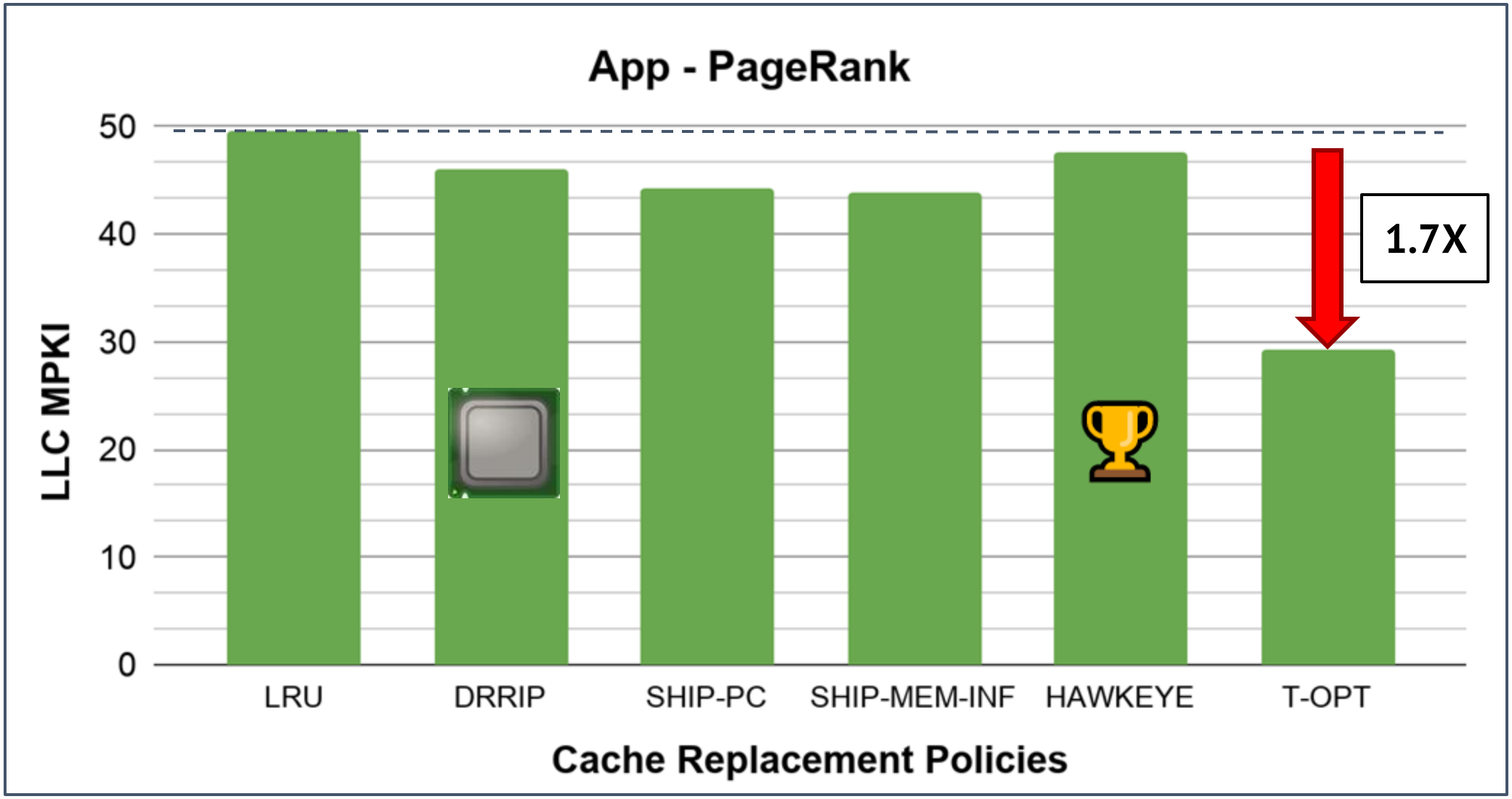
2-way Set-Associative

Which line should we evict?:

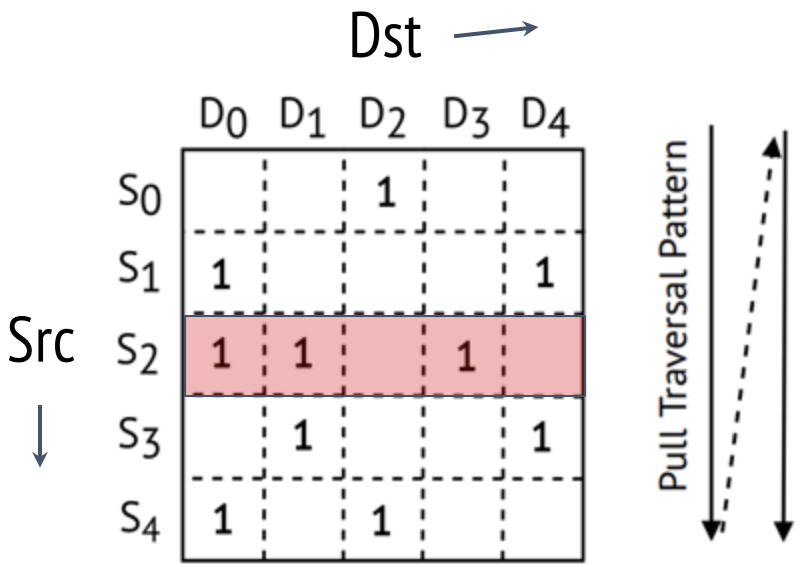
- srcData[S_1] **(nextRef @ D_4)** ✓
- srcData[S_2] **(nextRef @ D_1)**



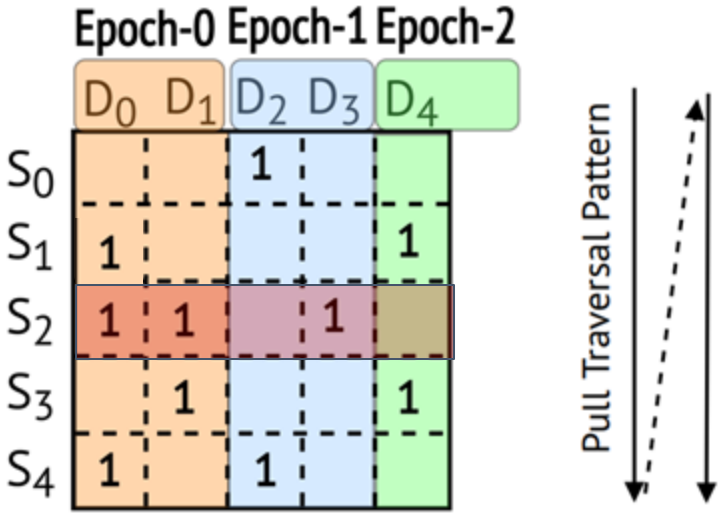
Transpose-based OPT (T-OPT) Provides Large Gains

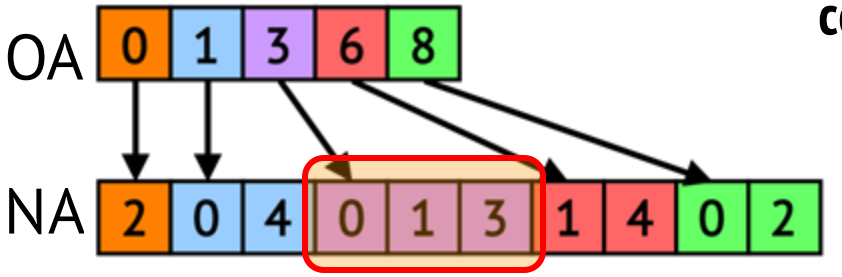


Main Technique: Use Quantization To Compress The Transpose



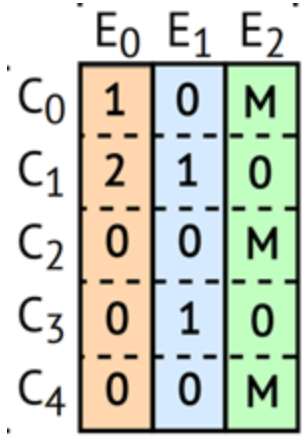
Divide execution into coarse-grained epochs





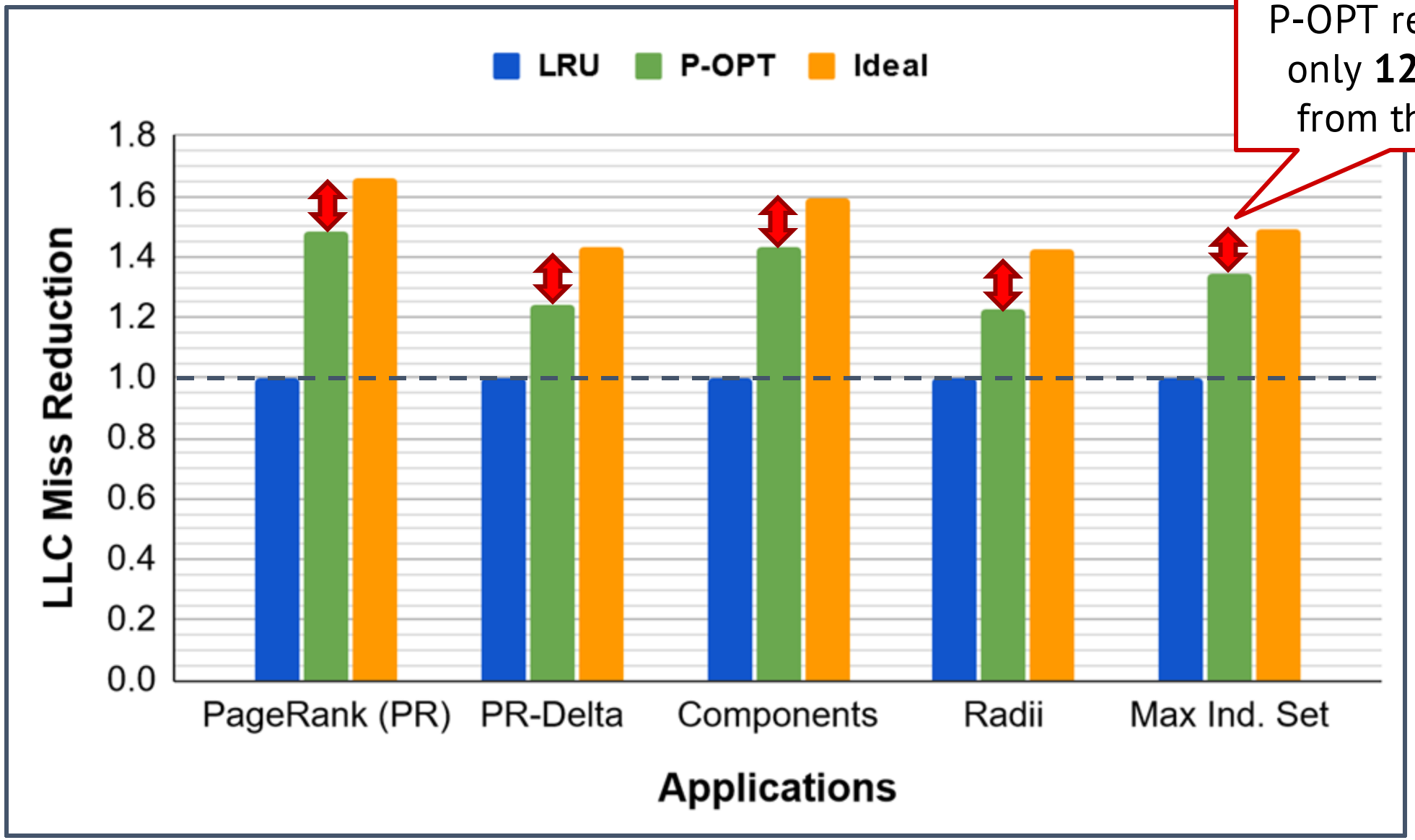
CSR
(Transpose)

Quantization enables compression of transpose data

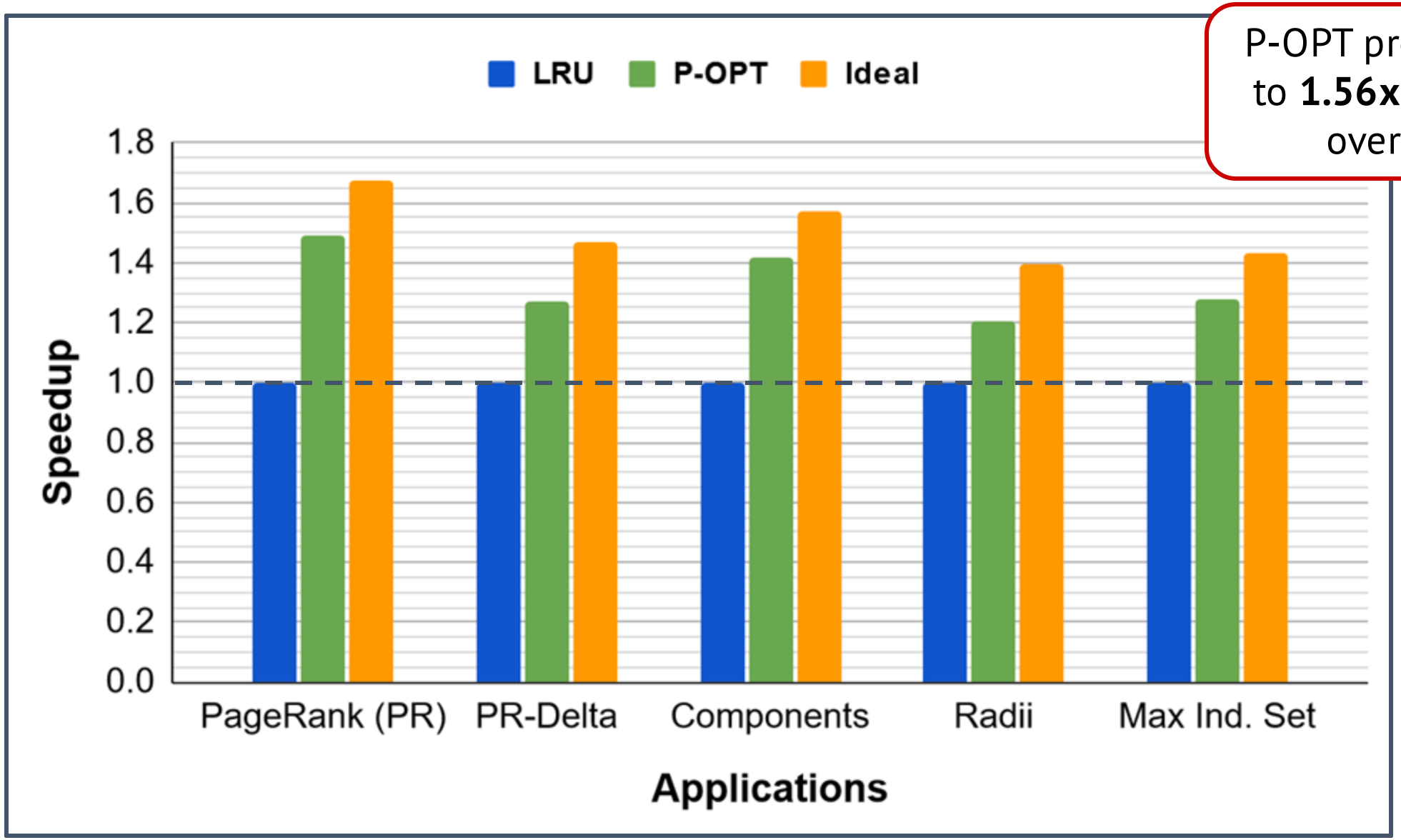


Rereference Matrix
(Quantized Transpose)

P-OPT Improves Cache Locality



P-OPT's LLC Miss Reductions Directly Translate To Speedups



P-OPT provides up to **1.56x** speedup over LRU

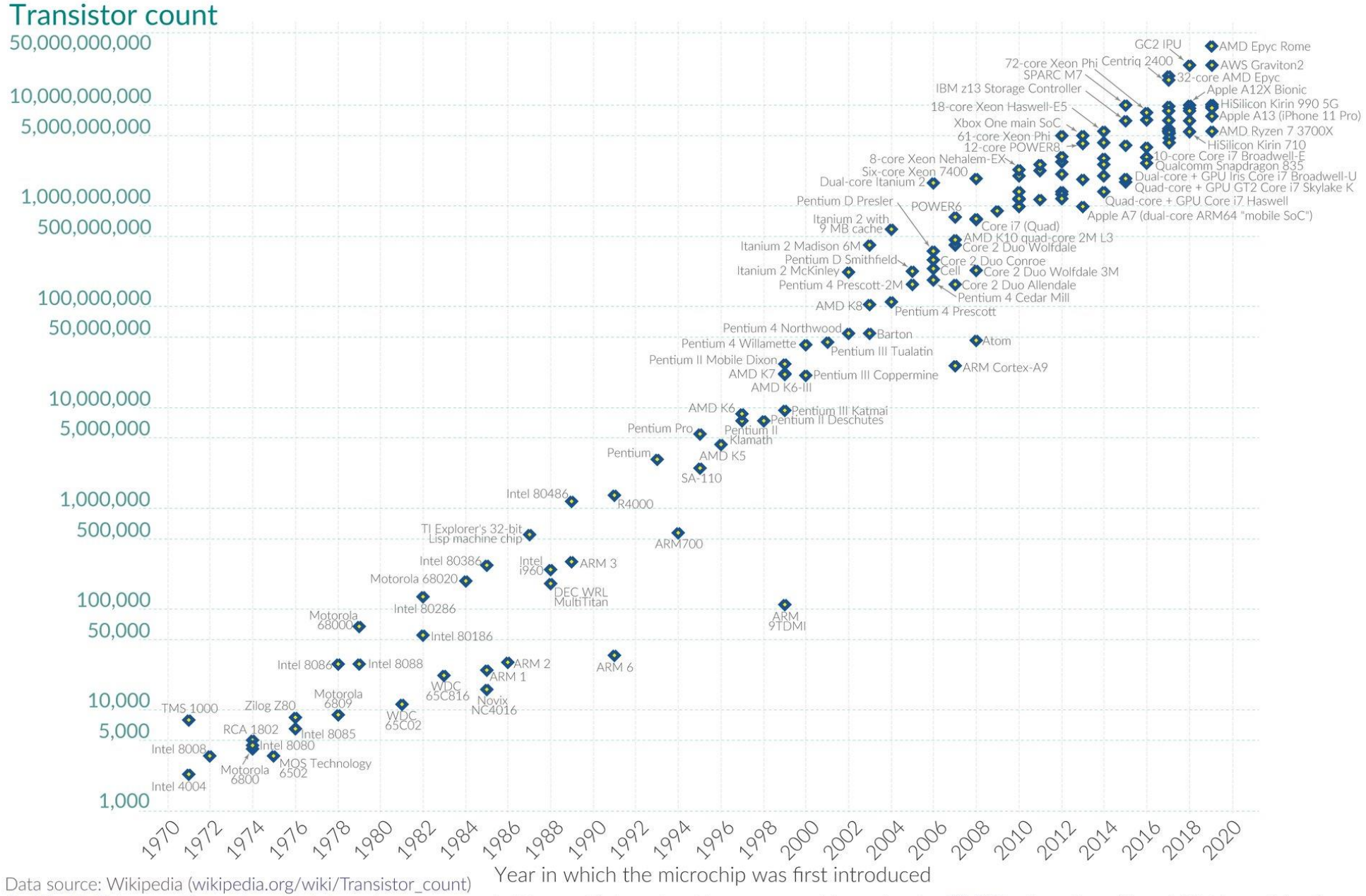
Today: Parallel Computer Architectures

- Why do we have mainly parallel computers
- How do we make caches work with parallelism
- Memory consistency models & ordering
- Implementing synchronization

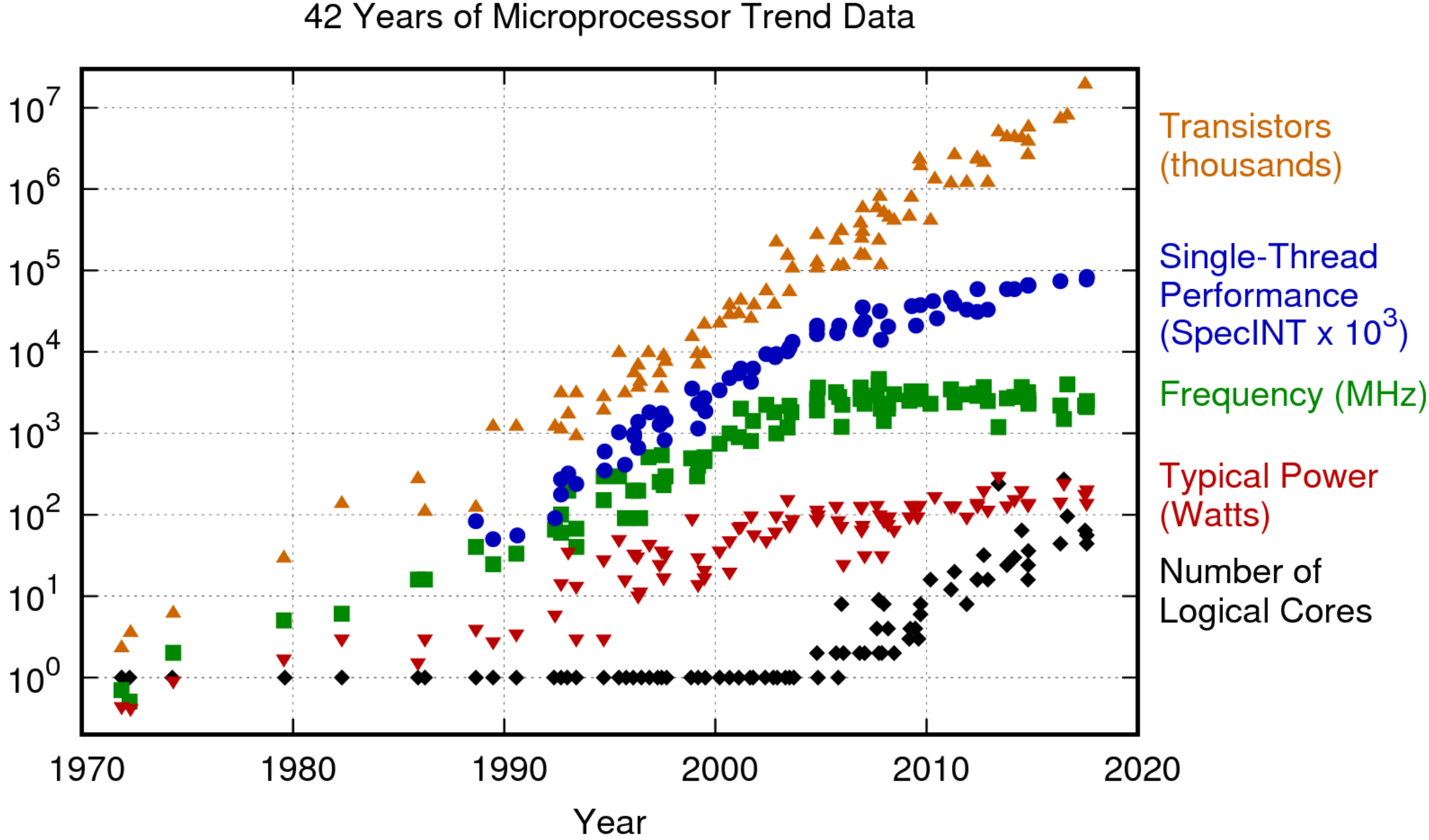
Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

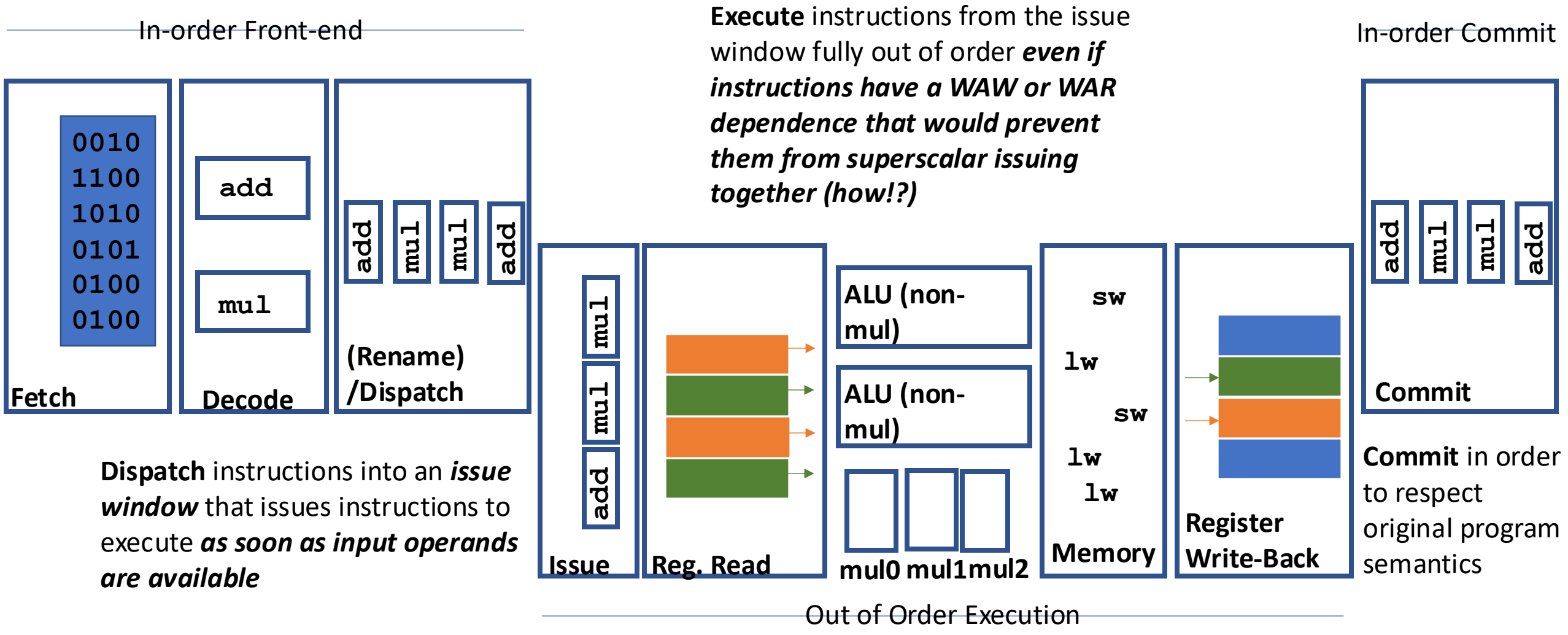


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

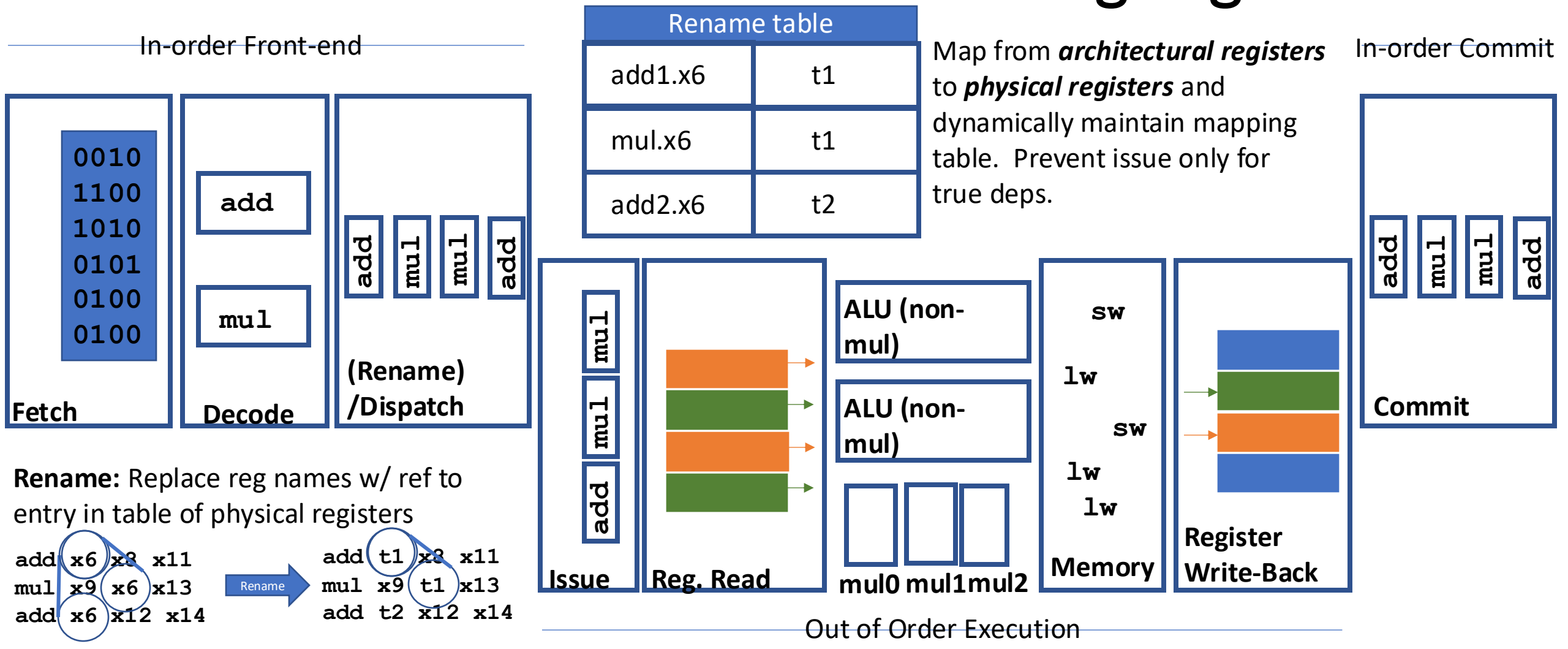


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Out of Order Execution



Register Renaming Resolves Dependences that Prevent Instructions from Executing Together

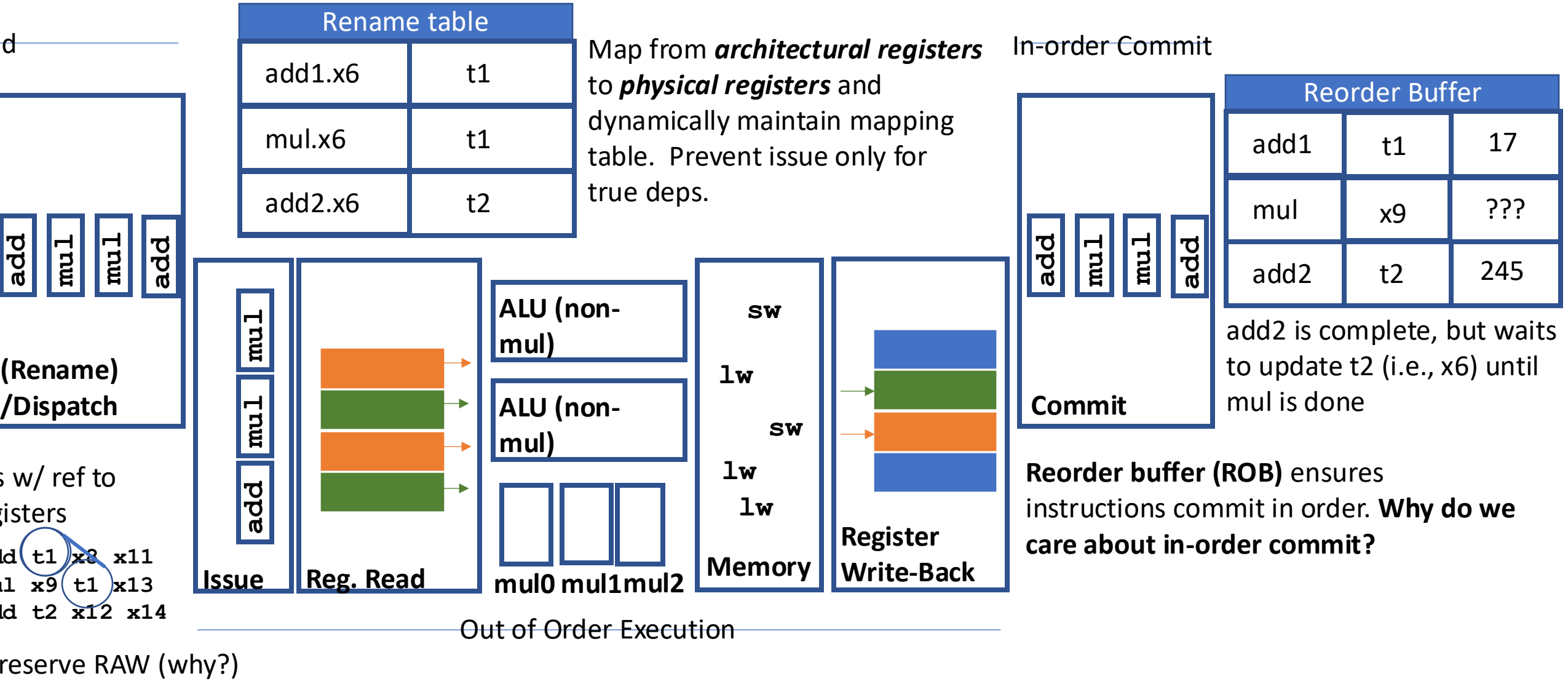


Rename: Replace reg names w/ ref to entry in table of physical registers

```
add x6  x8  x11          add t1  x8  x11
mul x9  x6  x13   Rename  mul x9  t1  x13
add x6  x12 x14          add t2  x12 x14
```

Eliminate WAW, WAR, and preserve RAW (why?)

In-order commit tracks instruction completion and ensures architectural state updates in order



Operand supply and commit scalability issues

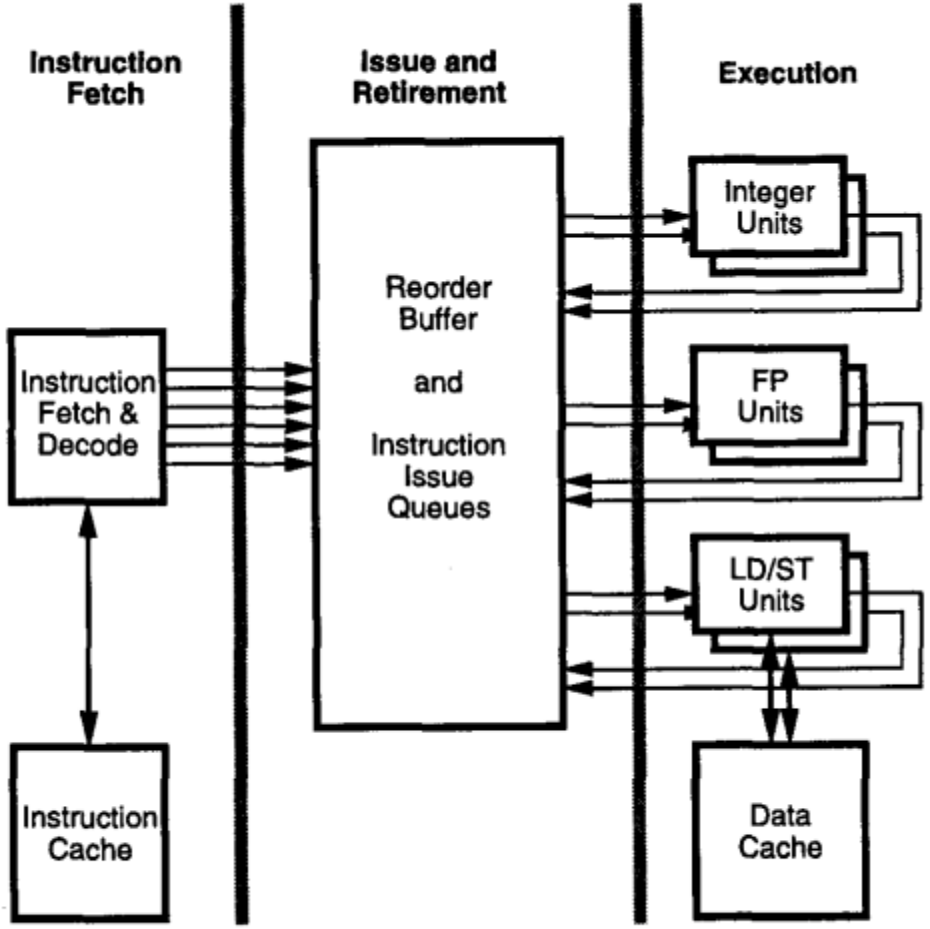


Figure 1. A dynamic superscalar CPU

The Case for a Single-Chip Multiprocessor

Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070
http://www-hydra.stanford.edu

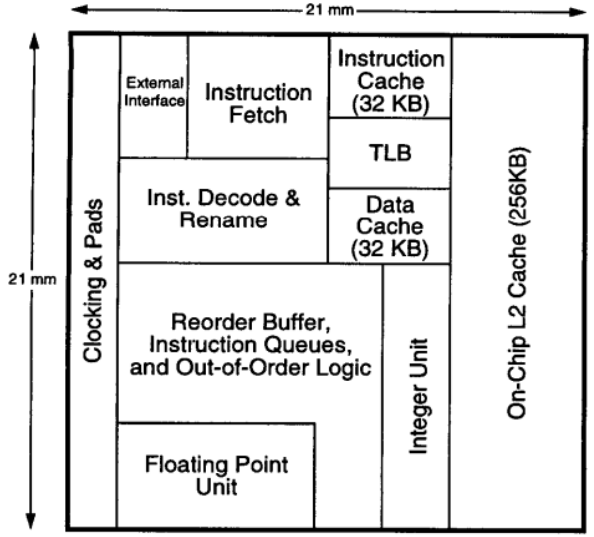


Figure 2. Floorplan for the six-issue dynamic superscalar microprocessor.

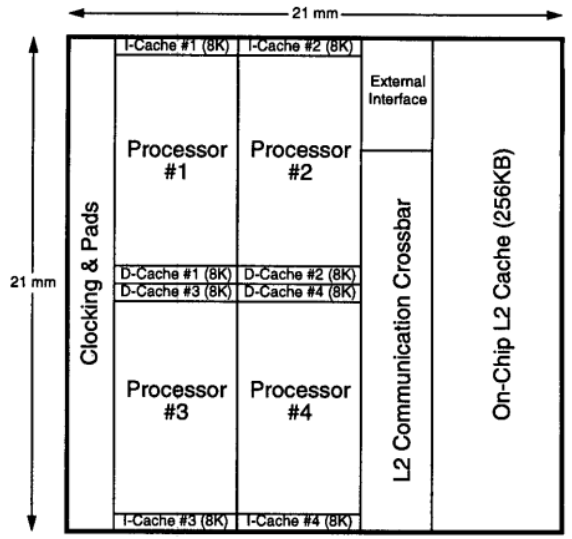


Figure 3. Floorplan for the four-way single-chip multiprocessor.

How do single-thread vs multi-thread chips scale?

CPU Component	0.35µm R10K Original Size (mm^2)	Size Extrapolated to 0.25µm (mm^2)	% Growth Due to New Functionality	New Size (mm^2)	% Area
256K On-Chip L2 Cache [a]	219	112	0%	112	26%
8-bank D Cache (32 KB)	26	13	25%	17	4%
8-bank I Cache (32 KB)	28	14	25%	18	4%
TLB Mechanism	10	5	200%	15	3%
External Interface Unit	27	14	0%	14	3%
Instruction Fetch Unit and BTB	18	9	200%	28	6%
Instruction Decode Section	21	11	250%	38	9%
Instruction Queues	28	14	250%	50	12%
Reorder Buffer	17	9	300%	34	9%
Integer Functional Units	20	10	200%	31	7%
FP Functional Units	24	12	200%	37	9%
Clocking & Overhead	73	37	0%	37	9%
Total Size	—	—	—	430	100%

Table 2. Size extrapolations for the 6-way superscalar from the MIPS R10000 processor

CPU Component	0.35µm R10K Original Size (mm^2)	Size Extrapolated to 0.25µm (mm^2)	% Growth Due to New Functionality	New Size (mm^2)	% Area (of CPU / of entire chip)
D Cache (8 KB)	26	13	-75%	3	6% /3%
I Cache (8 KB)	28	14	-75%	4	7% / 3%
TLB Mechanism	10	5	0%	5	9% / 5%
Instruction Fetch Unit and BTB	18	9	-25%	7	13% / 7%
Instruction Decode Section	21	11	-50%	5	10% / 5%
Instruction Queues	28	14	-70%	4	8% / 4%
Reorder Buffer	17	9	-80%	2	3% / 2%
Integer Functional Units	20	10	0%	10	20% / 10%
FP Functional Units	24	12	0%	12	23% / 12%
Per-CPU Subtotal	—	—	—	53	100% / 50%
256K On-Chip L2 Cache [a]	219	112	0%	112	26%
External Interface Unit	27	14	0%	14	3%
Crossbar Between CPUs	—	—	—	50	12%
Clocking & Overhead	73	37	0%	37	9%
Total Size	—	—	—	424	100%

Table 3. Size extrapolations in the 4 × 2-way MP from the MIPS R10000 processor.

Performance of single- vs. multi-threaded

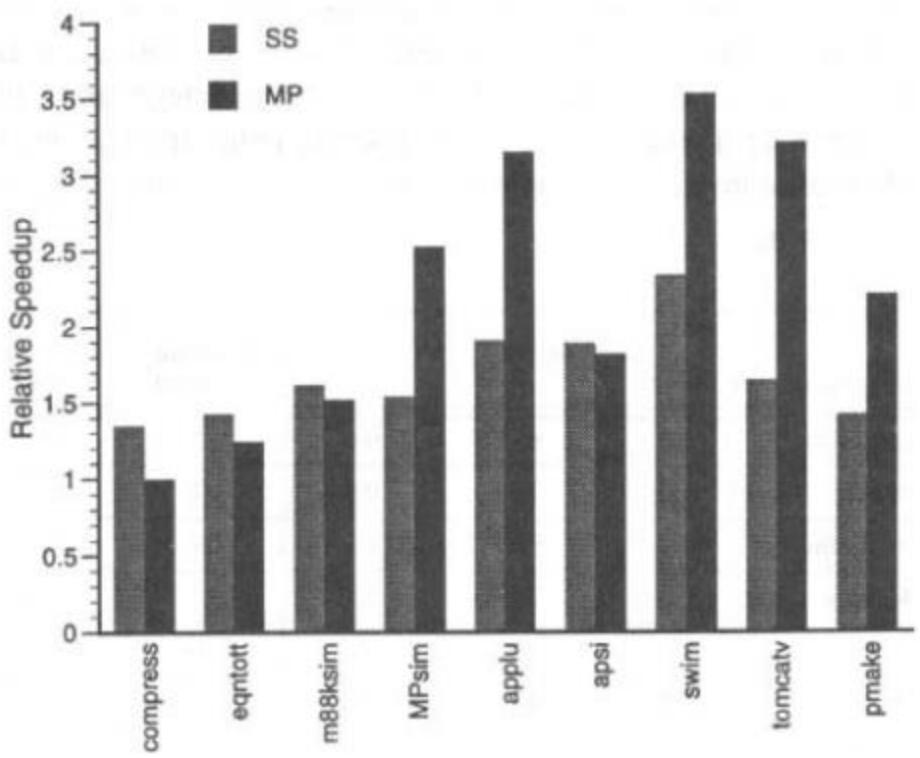
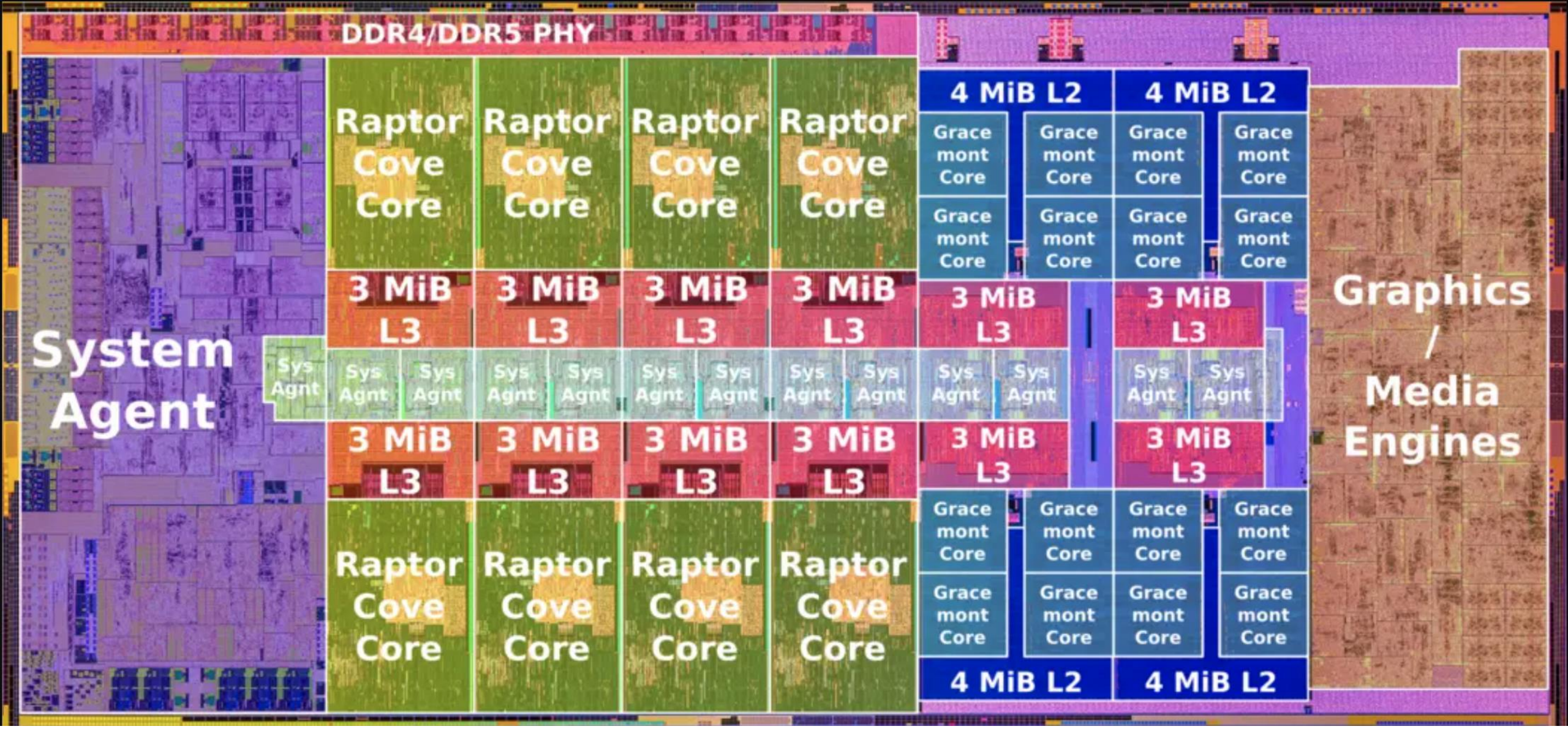
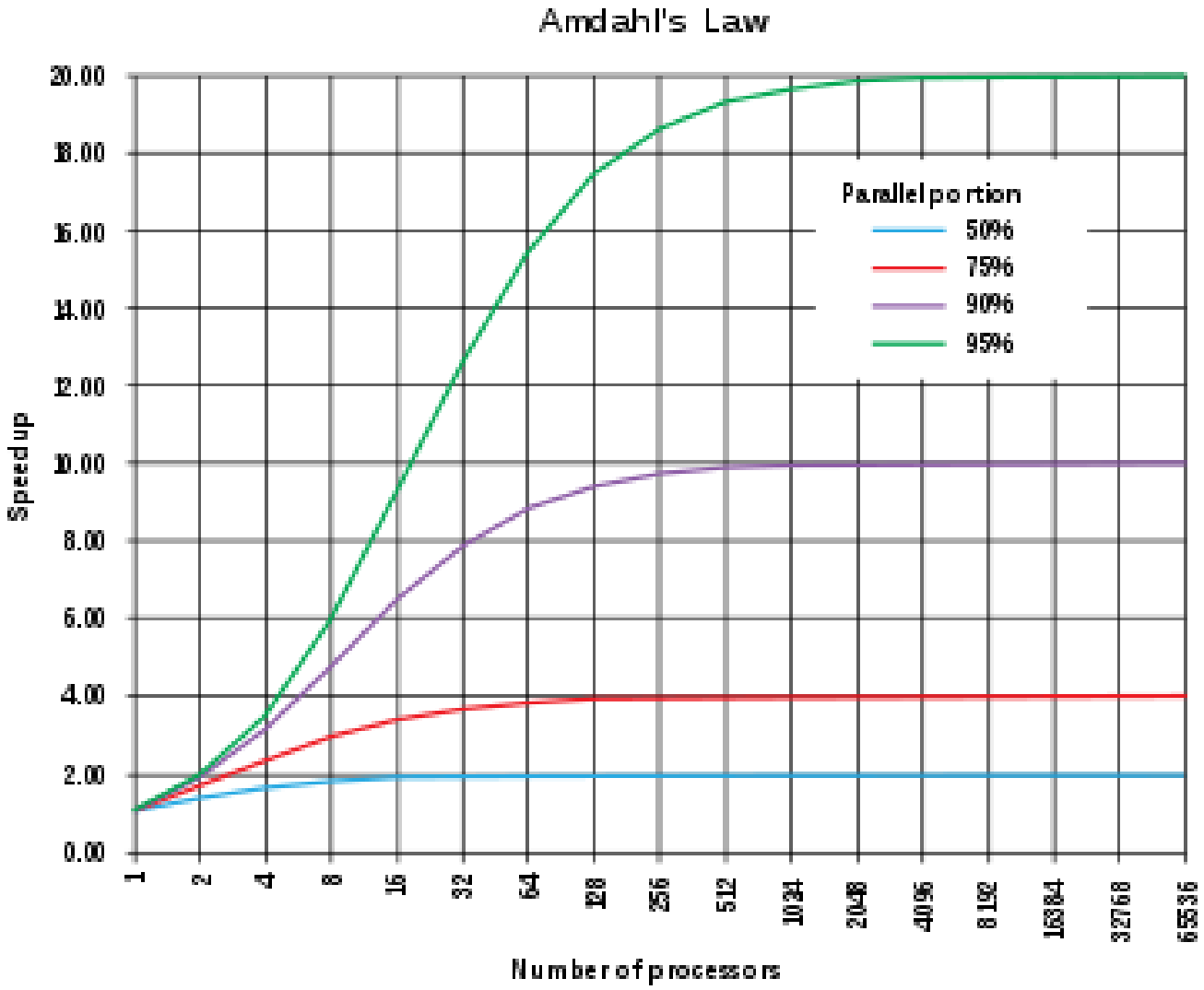


Figure 6. Performance comparison of SS and MP.

- 7nm process
- 17 metal layers
- ~25.9B transistors
- ~257 mm² die size
- 8 Perf Cores + 16 Efficiency Cores + GPU



Shared memory multi-threading



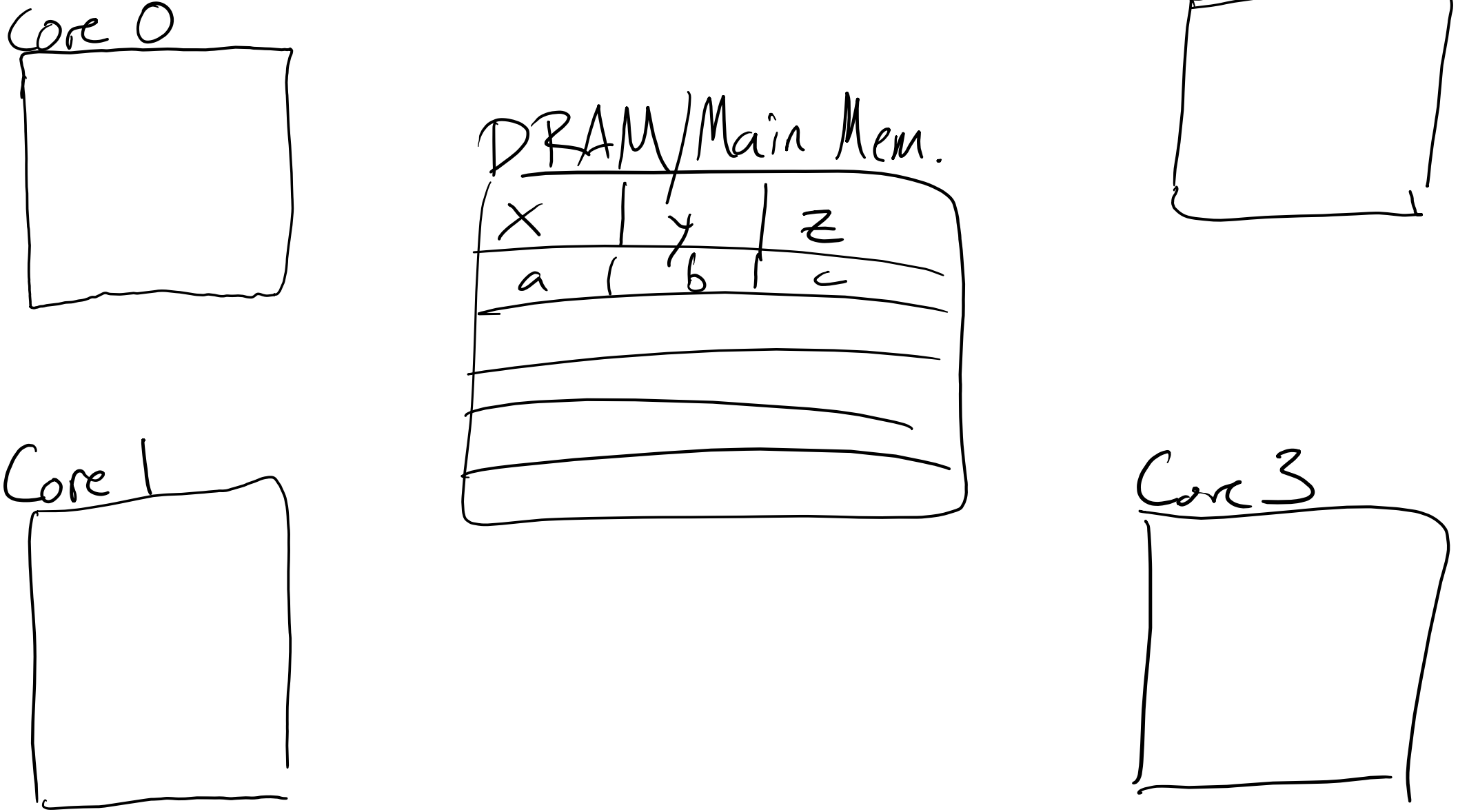
Parallel hardware + parallelizable software are a direct application of Amdahl's Law

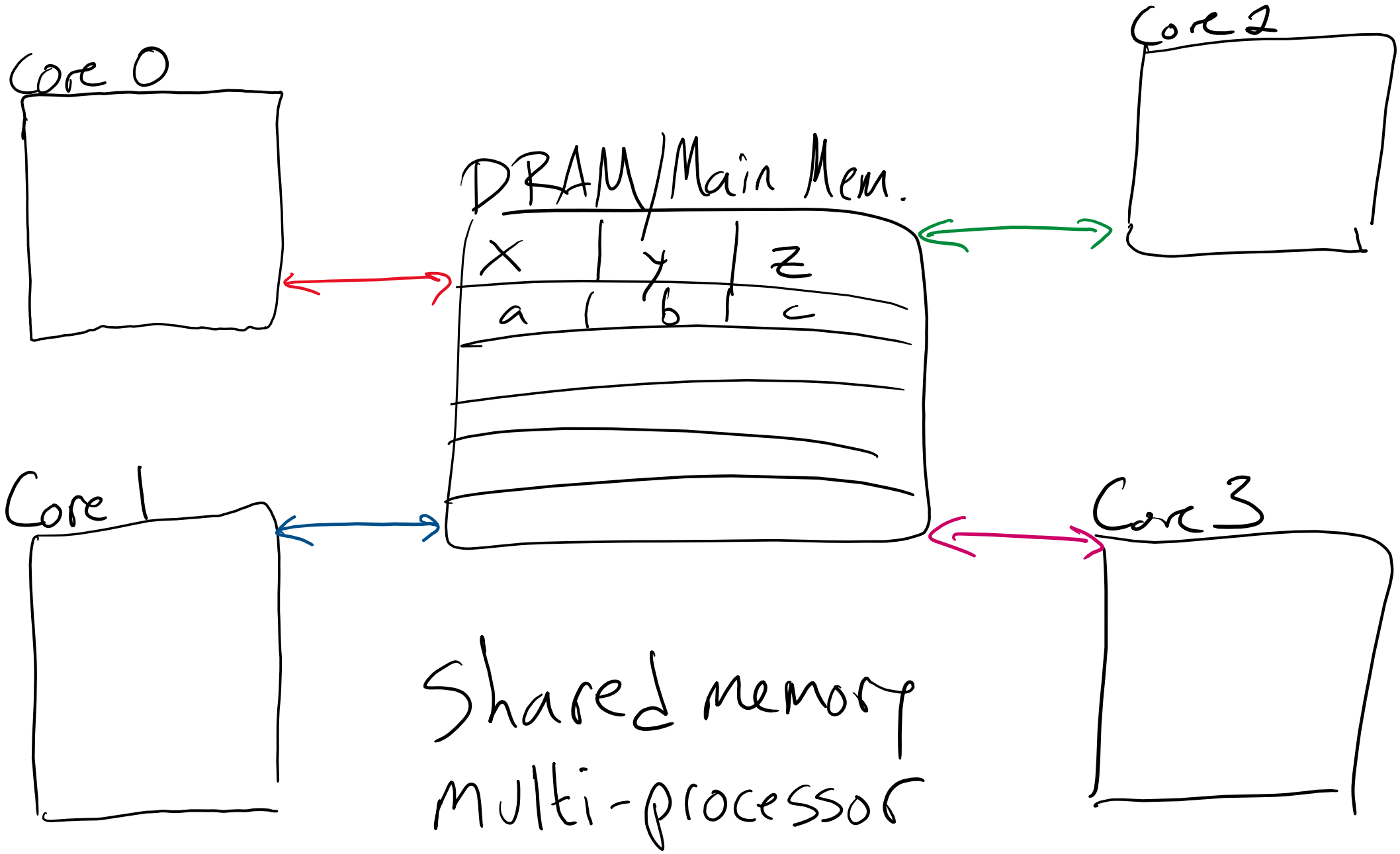
Multi-core parallelism was the primary way to keep performance scaling alive once single-thread performance hit the wall

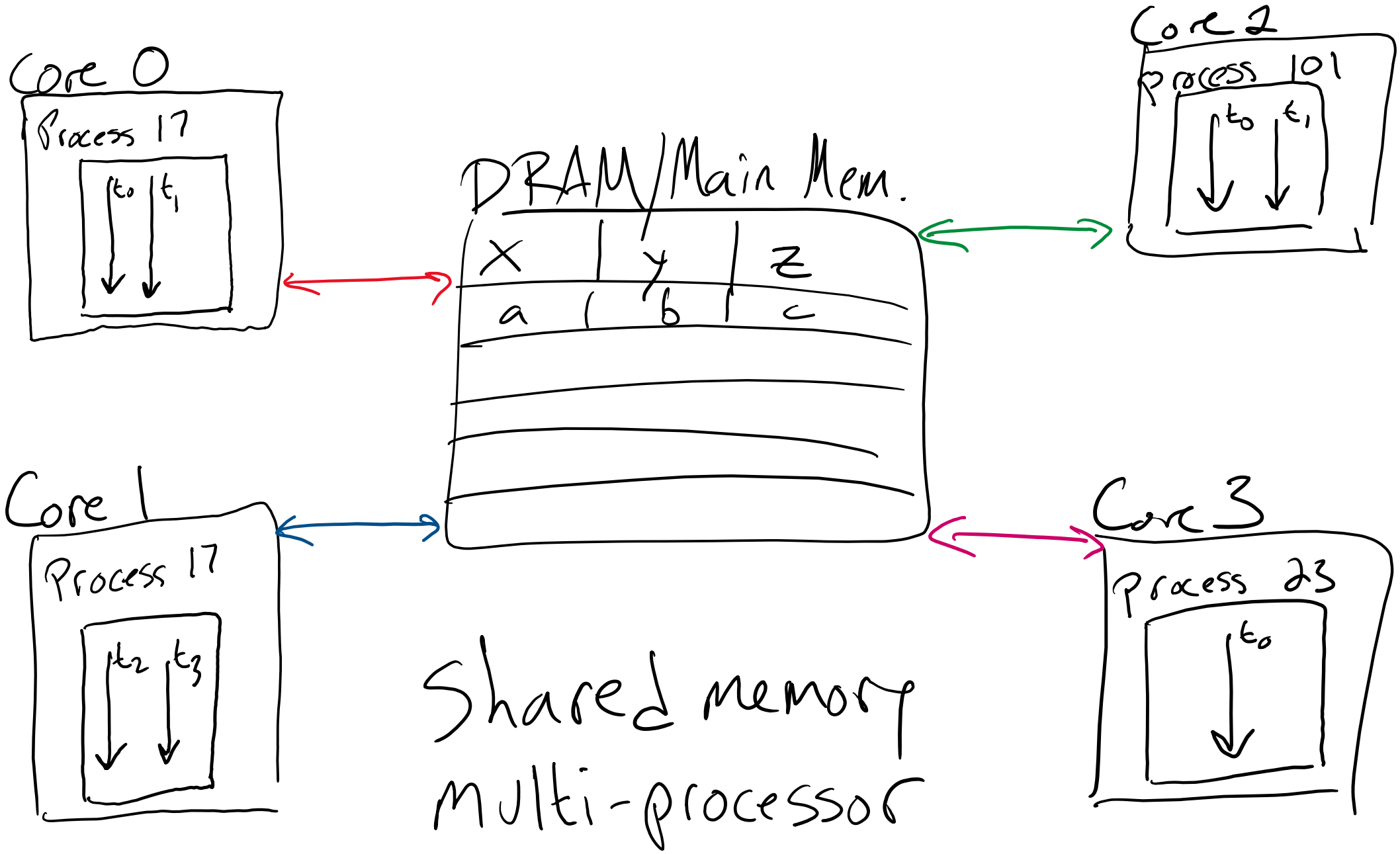
How to we architect a *programmable* parallel computer system?

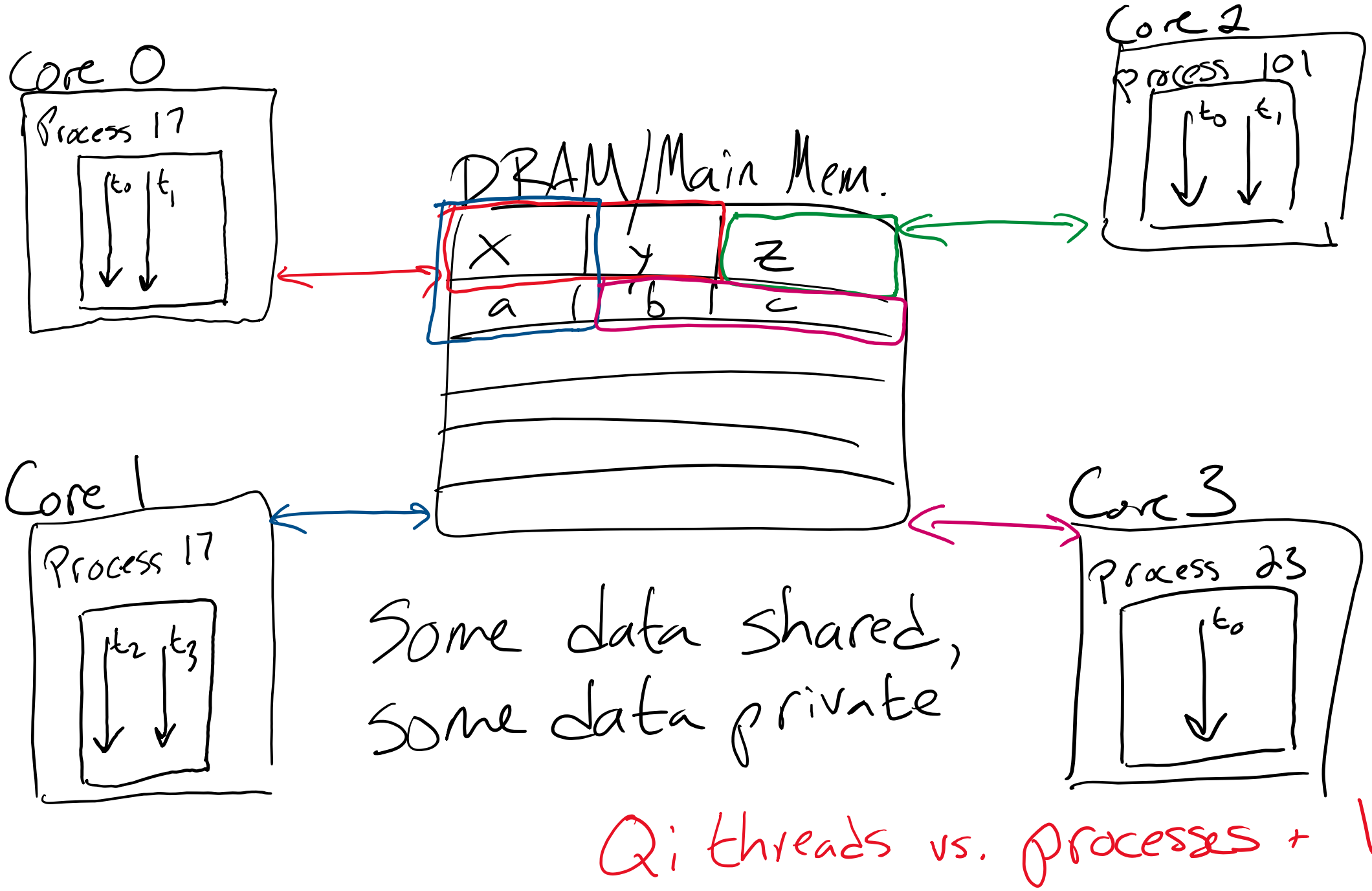
What are the main impediments to parallel programmability?

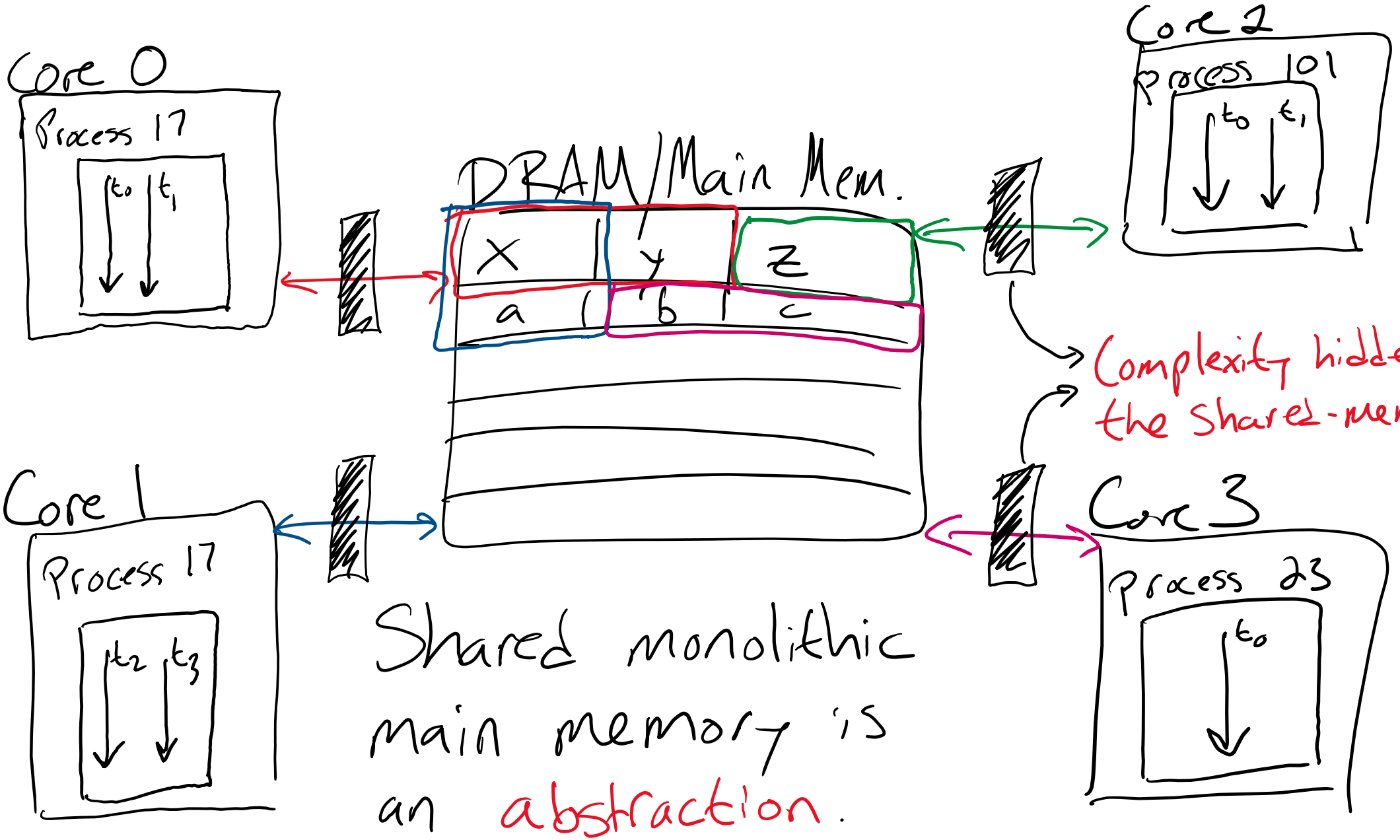
To parallel optimization?

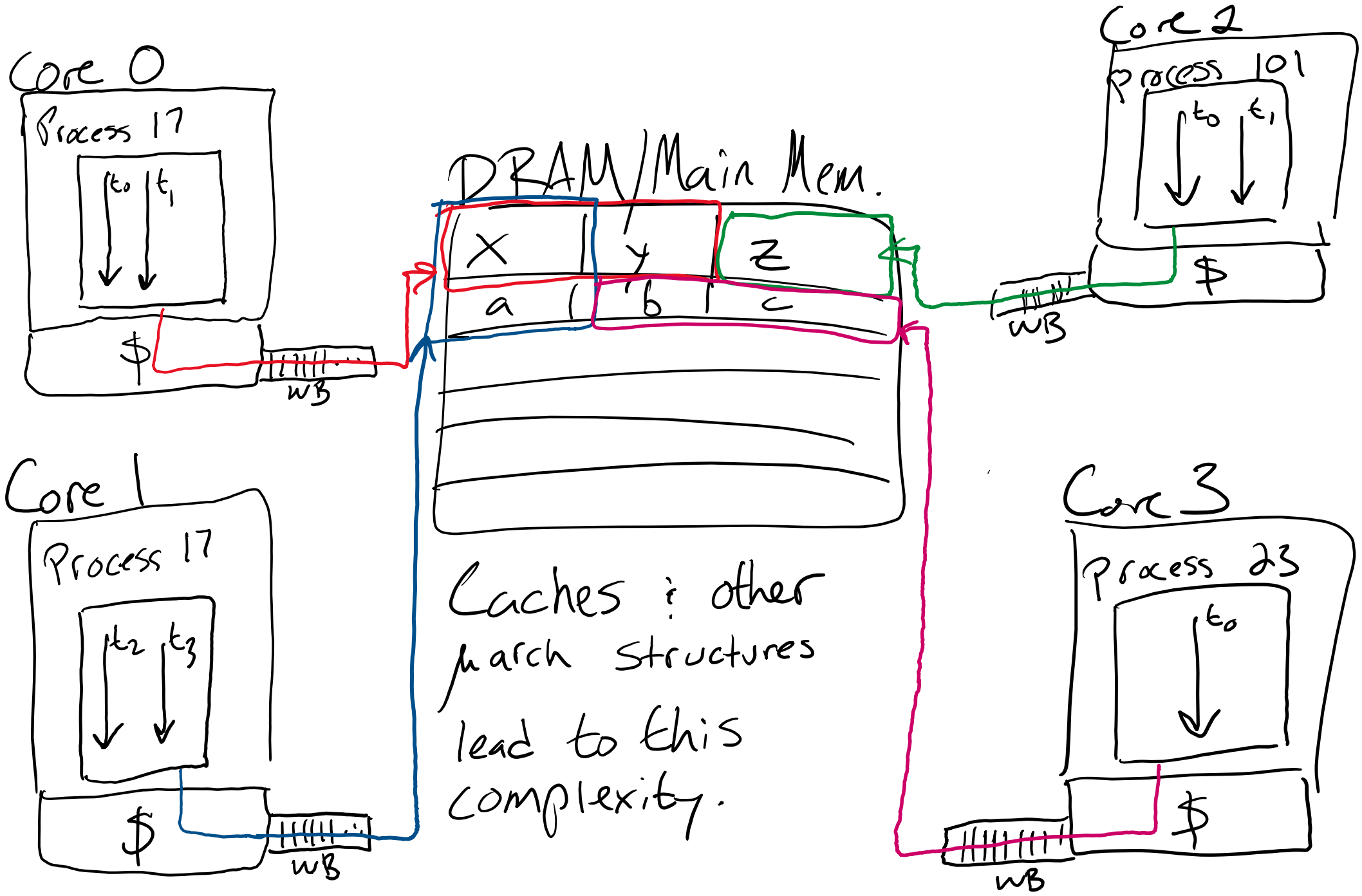

Core 0
Core 2
DRAM/Main Mem.
x | y | z
a | b | c
Core 1
Core 3


Core 0
Core 2
DRAM/Main Mem.
x
y
z
a
b
c
Core 1
Core 3
Shared memory
multi-processor


Core 0
Process 17
t_0
t_1
DRAM/Main Mem.
X
y
z
a
b
c
Core 2
Process 101
t_0
t_1
Core 1
Process 17
t_2
t_3
Core 3
Process 23
t_0
Shared memory
multi-processor


Core 0
Process 17
t_0
t_1
DRAM/Main Mem.
x
y
z
a
b
c
Core 2
process 101
t_0
t_1
Core 1
Process 17
t_2
t_3
Core 3
process 23
t_0
Some data shared,
some data private
Q: threads vs. processes + VM?

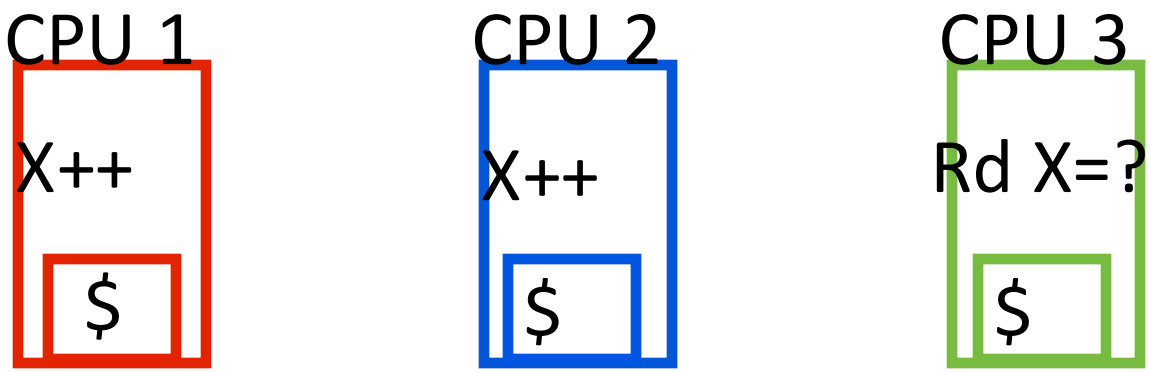

Core 0
Process 17
t_0
t_1
DRAM/Main Mem.
x
y
z
a
b
c
Core 2
Process 101
t_0
t_1
Complexity hidden by the shared-mem abstraction
Core 1
Process 17
t_2
t_3
Core 3
Process 23
t_0
Shared monolithic main memory is an abstraction.


Core 0
Process 17
t_0
t_1
$
WB
DRAM/Main Mem.
X
y
z
a
b
c
Core 2
Process 101
t_0
t_1
$
WB
Core 1
Process 17
t_2
t_3
$
WB
Caches & other
march structures
lead to this
complexity.
Core 3
Process 23
t_0
$
WB

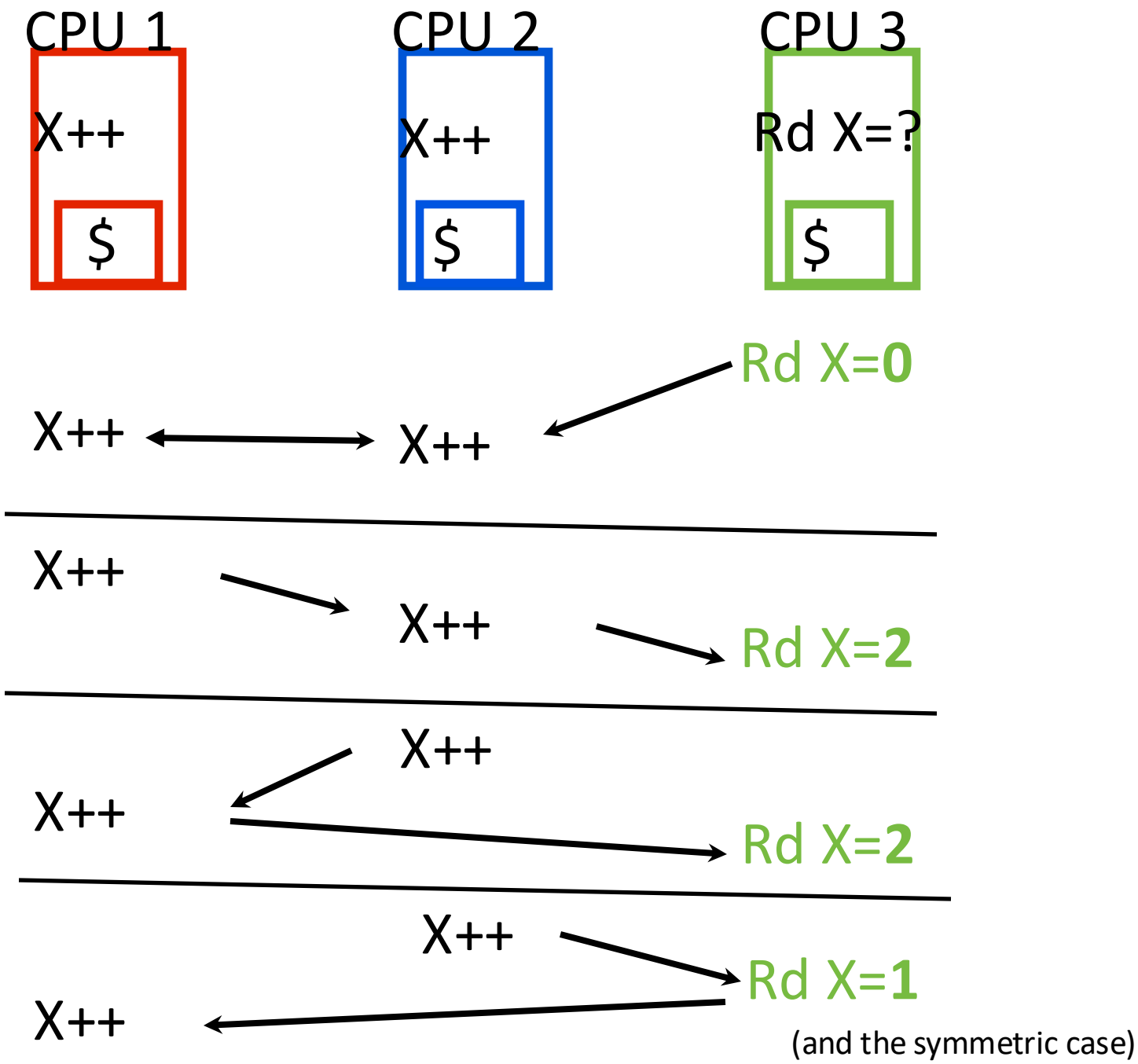
"Coherence seeks to **make the caches of a shared-memory system as functionally invisible as the caches in a single-core system**. Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores."

Excerpt from "Primer on Memory Consistency and Cache Coherence"
Mark Hill, 2011

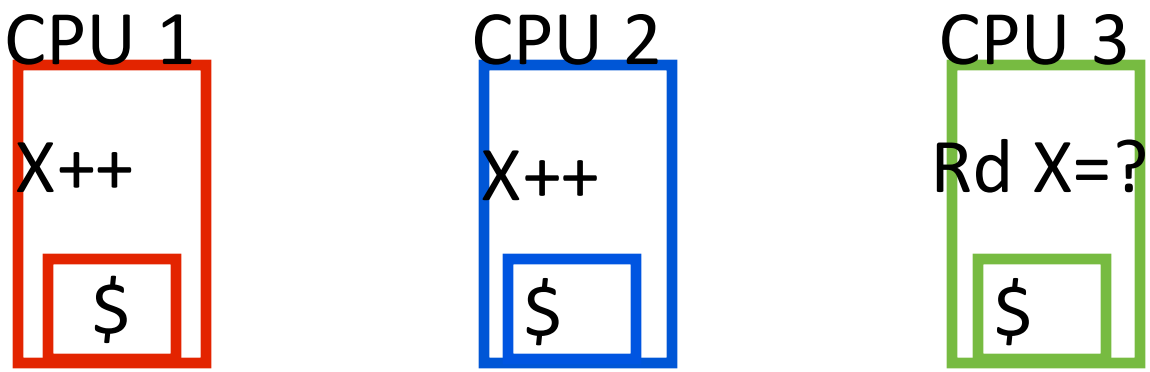
Cache Coherence



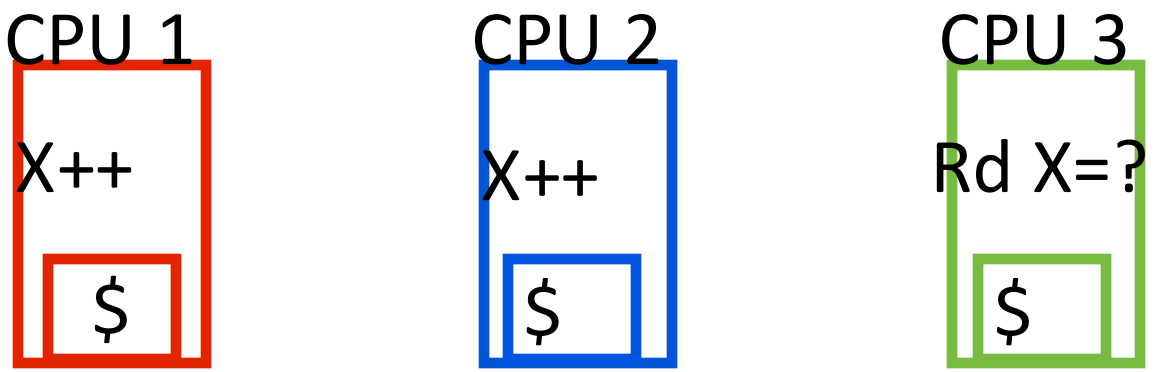
What is the behavior of this parallel program?
(X initially 0)



(and the symmetric case)

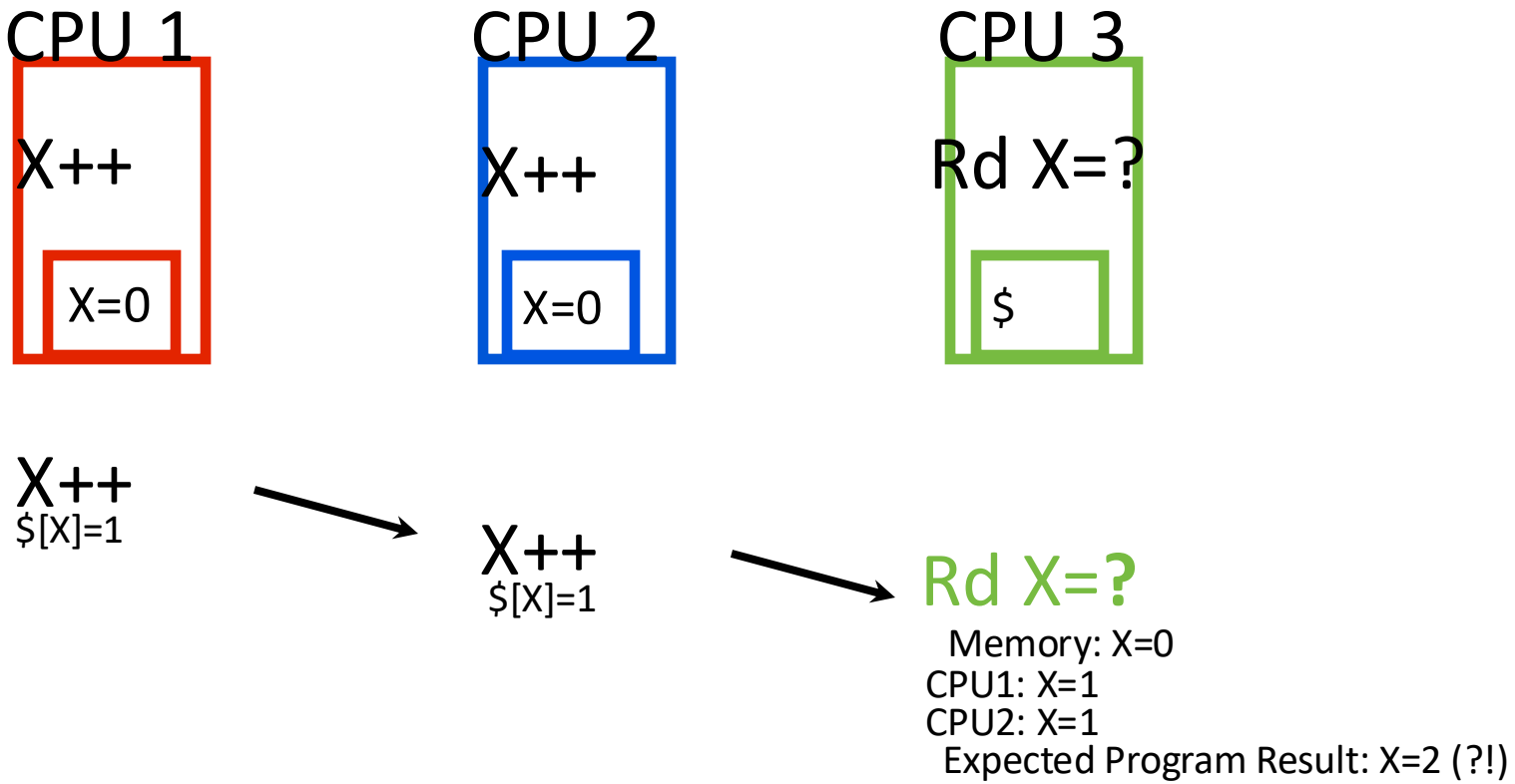


What assumptions are we making about the system to produce the results 0, 1, and 2?

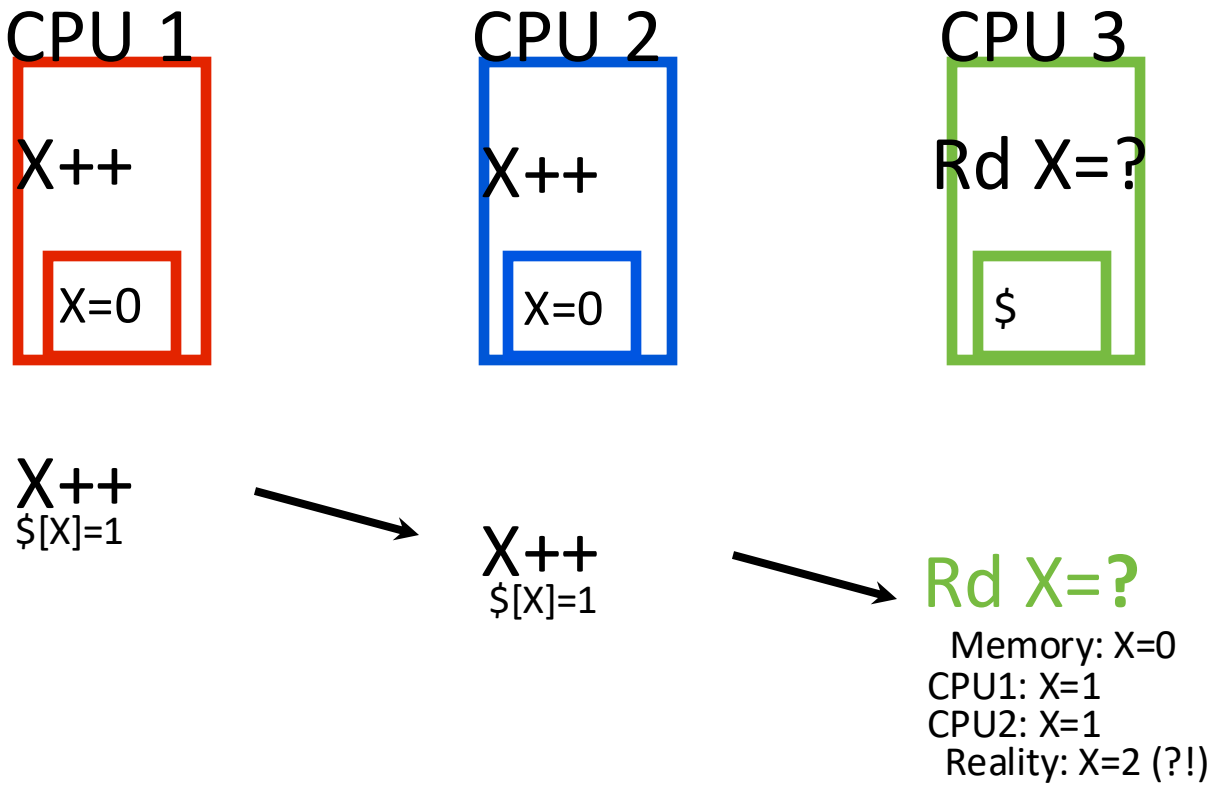


We assume the updates see one anothers' results!

Q: Why wouldn't they see one anothers' updates?


CPU 1
X++
X=0
CPU 2
X++
X=0
CPU 3
Rd X=?
$
X++
$[X]=1
X++
$[X]=1
Rd X=?
Memory: X=0
CPU1: X=1
CPU2: X=1
Expected Program Result: X=2 (?!)

What now?



Never let this happen. Caches should be **coherent.**

"coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores"

Defining Coherence

"Coherence serializes all reads with all updates to the same location by different CPUs/caches, so that each read sees the result of the most recent update by any other"

From the Primer on Cache Coherence & Consistency

Single Writer/Multiple Reader (SWMR) Invariant:

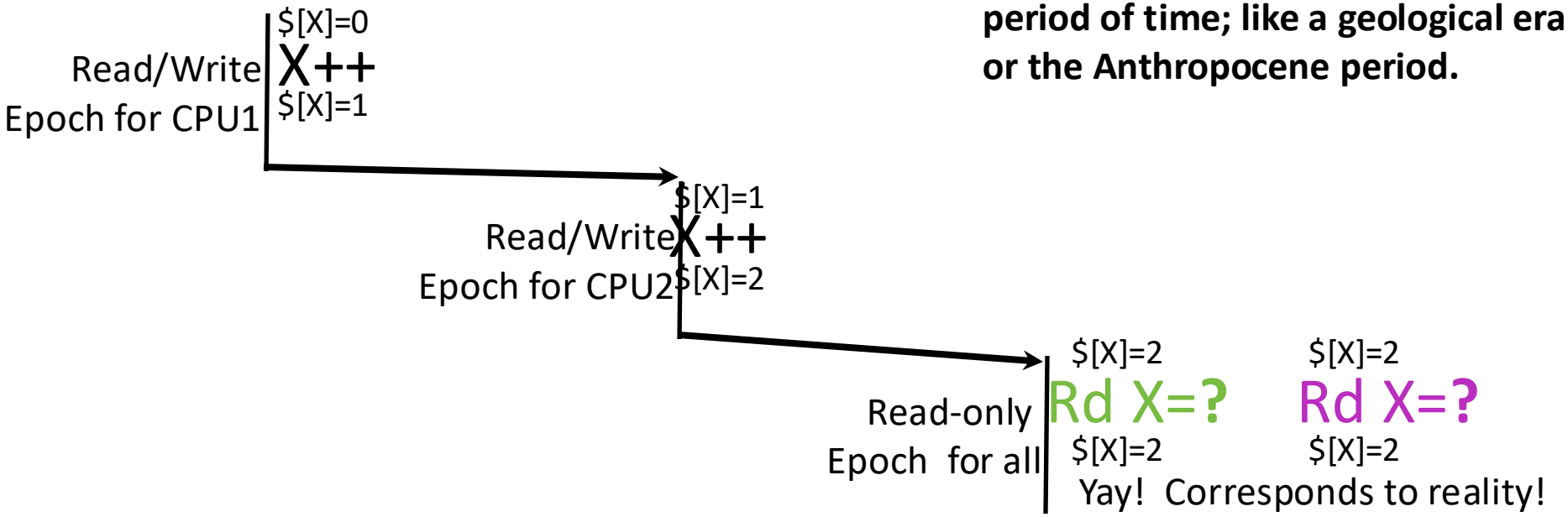
There is *one writer* or an *arbitrary number of readers* of a block of memory that could be cached at any given time.

Data-Value Invariant:

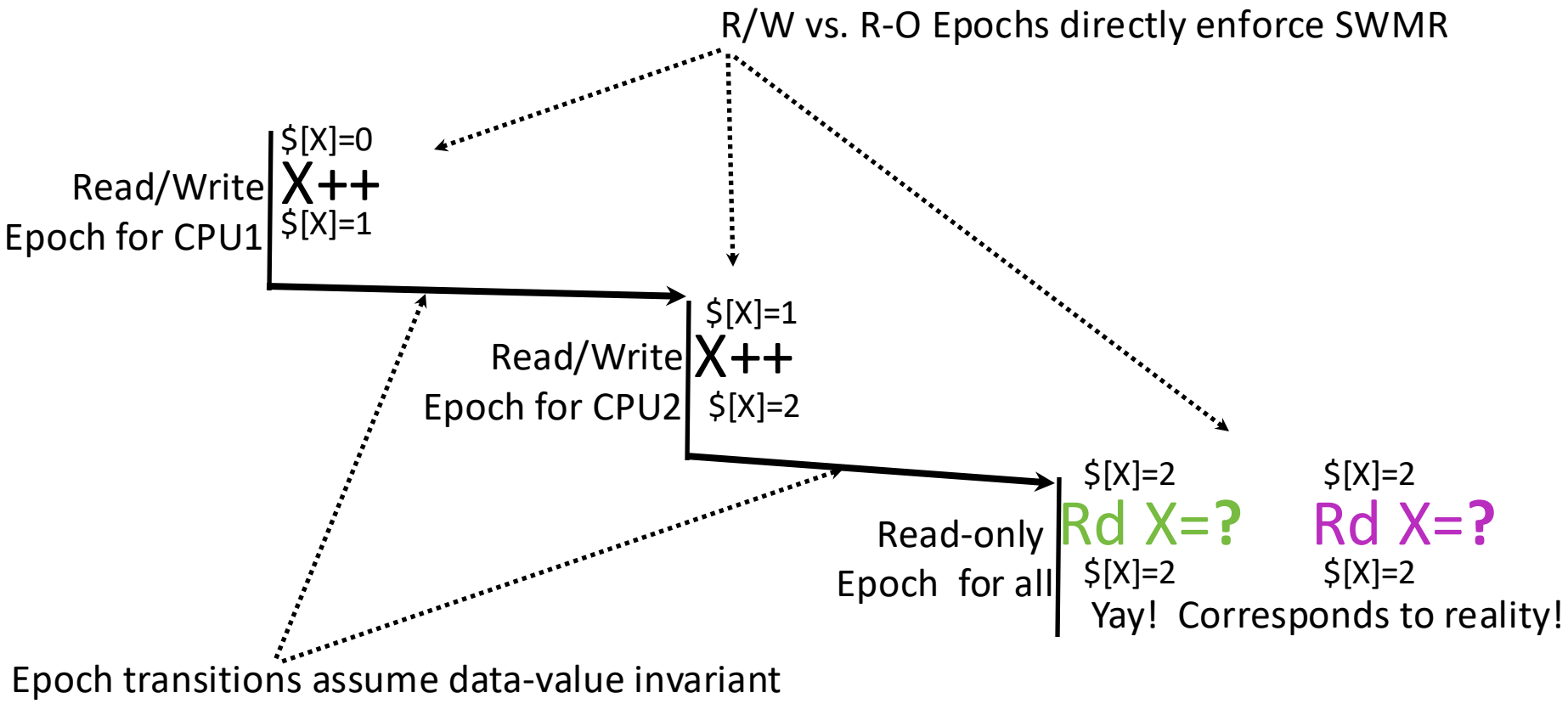
There is a globally defined *most recent write* and a read always reads the value written by the most recent write before that read

Epoch Model

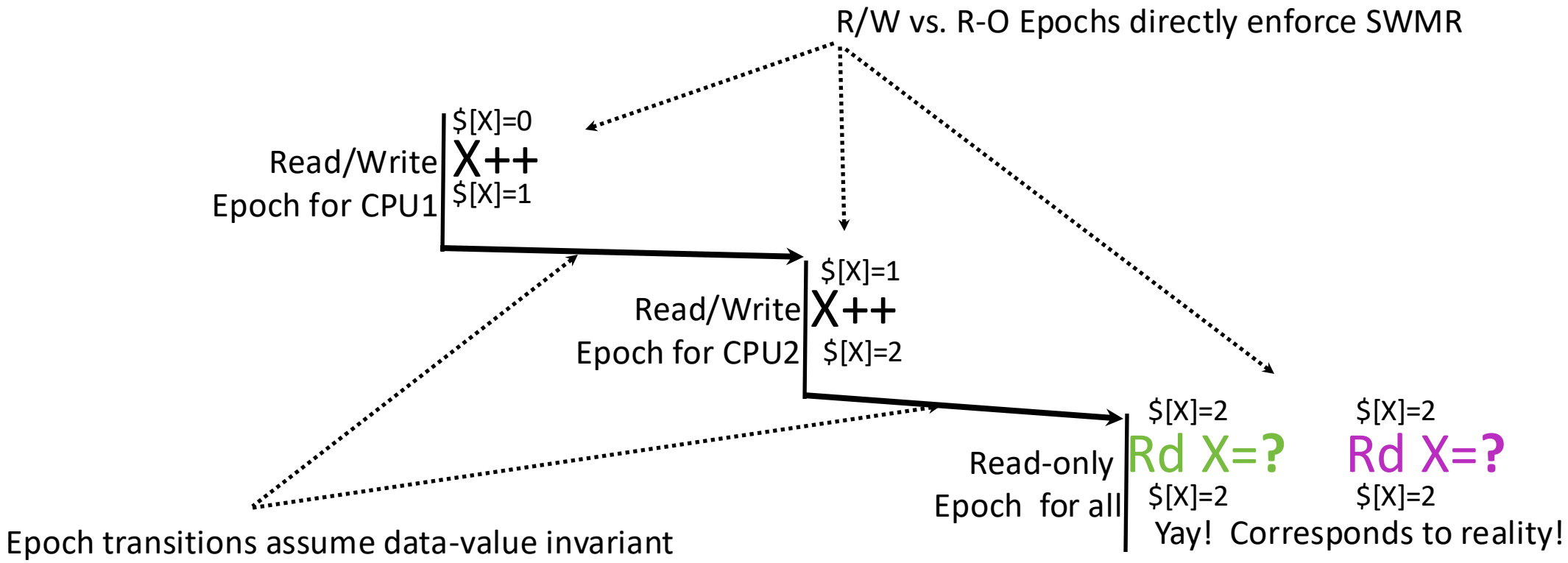
Definition: an "epoch" is a period of time the extents of which are determined by a property of that period of time; like a geological era or the Anthropocene period.



Epoch Model


R/W vs. R-O Epochs directly enforce SWMR
$[X]=0
Read/Write
X++
Epoch for CPU1
$[X]=1
$[X]=1
Read/Write
X++
Epoch for CPU2
$[X]=2
$[X]=2
$[X]=2
Read-only
Rd X=?
Rd X=?
Epoch for all
$[X]=2
$[X]=2
Yay! Corresponds to reality!
Epoch transitions assume data-value invariant

Epoch Model



Question: What are the requirements for an implementation of the Epoch Model?

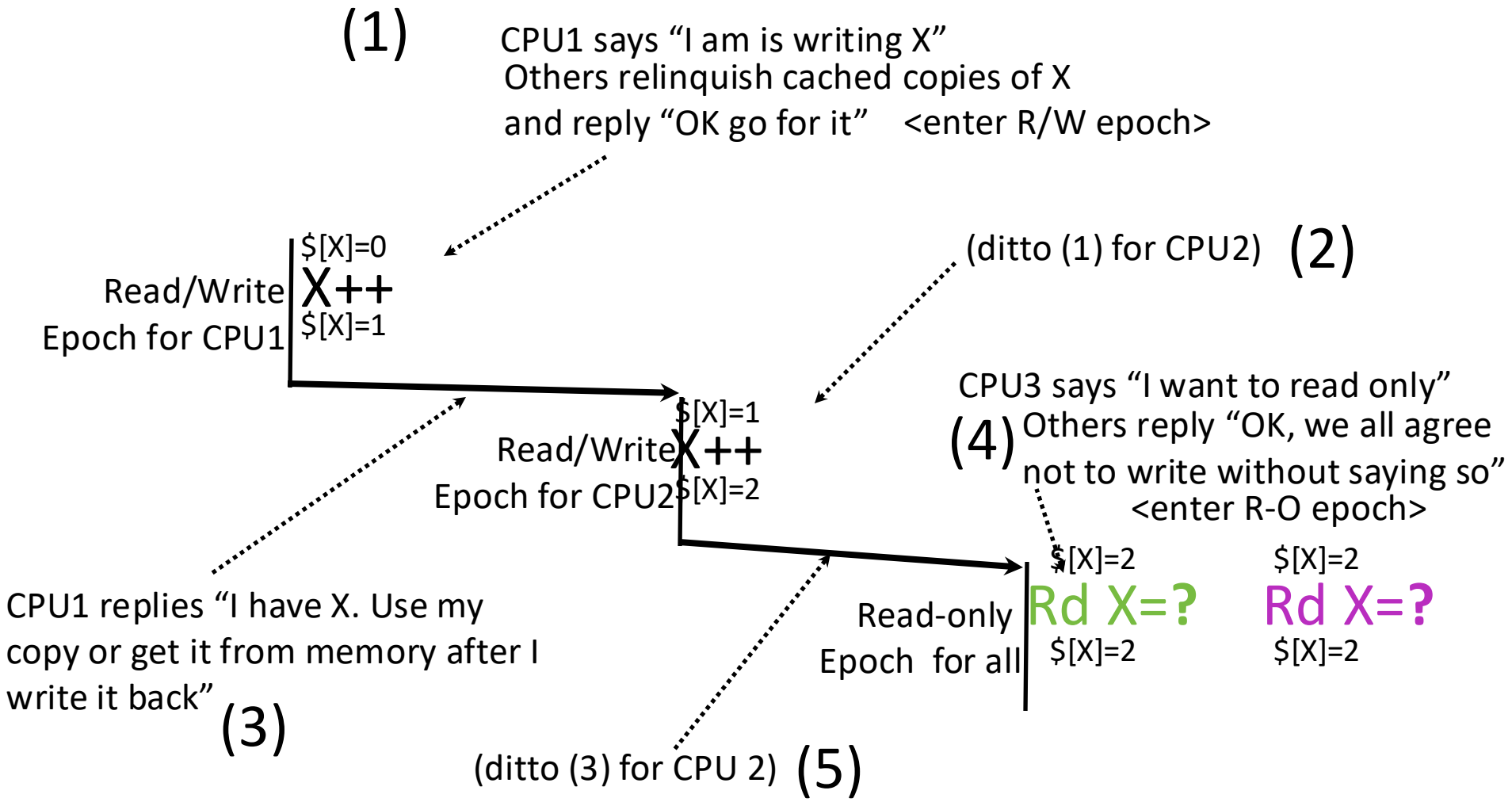
Cache Coherence Protocol

Add state to each cache line saying whether it is R-O or R/W

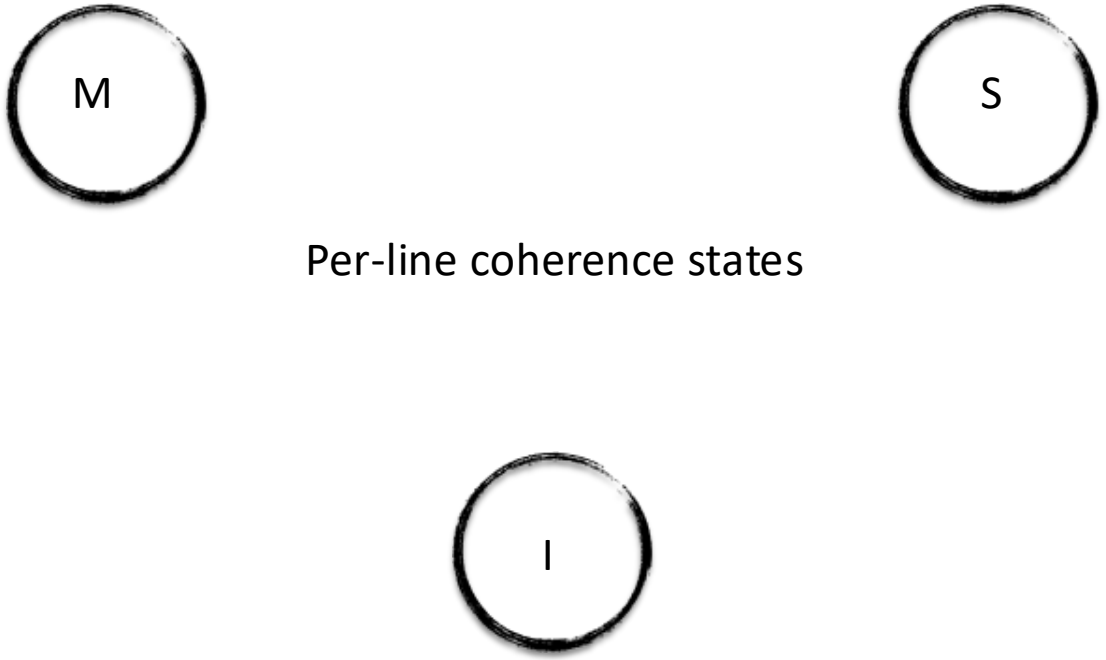
Add protocol actions to move lines from state to state based on (1)local memory operations; and (2)other CPUs' memory operations

Add support to get data from (1)local cache; (2)a remote cache; or (3)main memory, depending on line's protocol state

High-level sketch of protocol in action

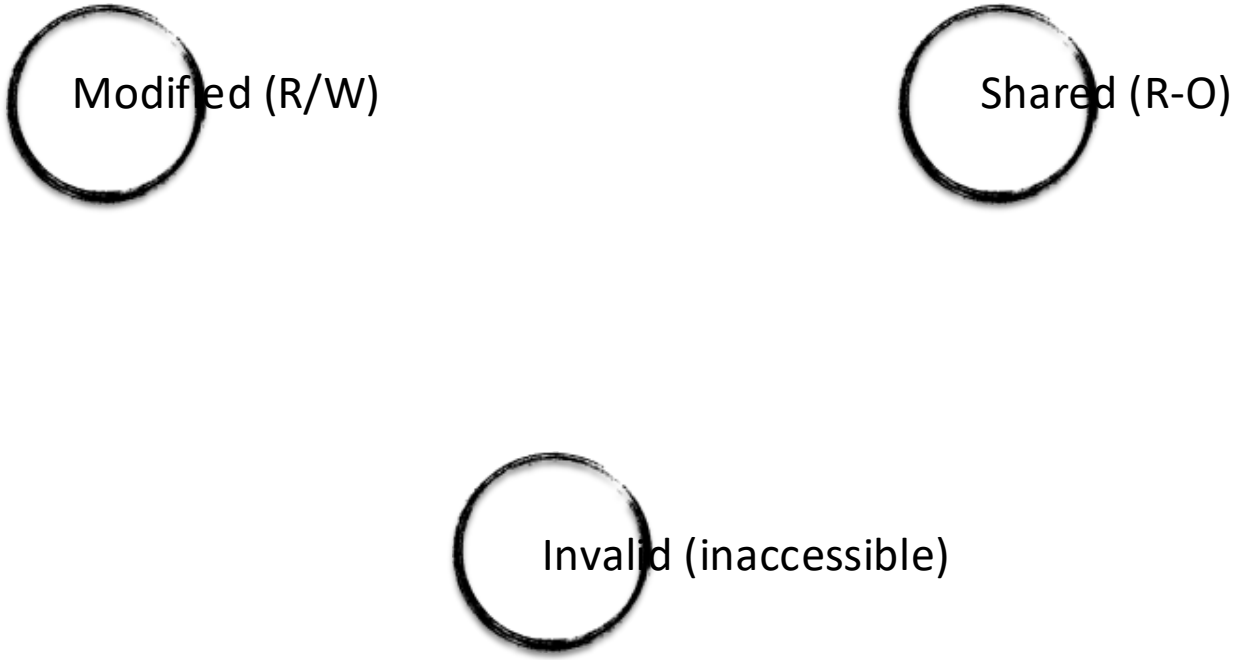


Cache Coherence Protocol



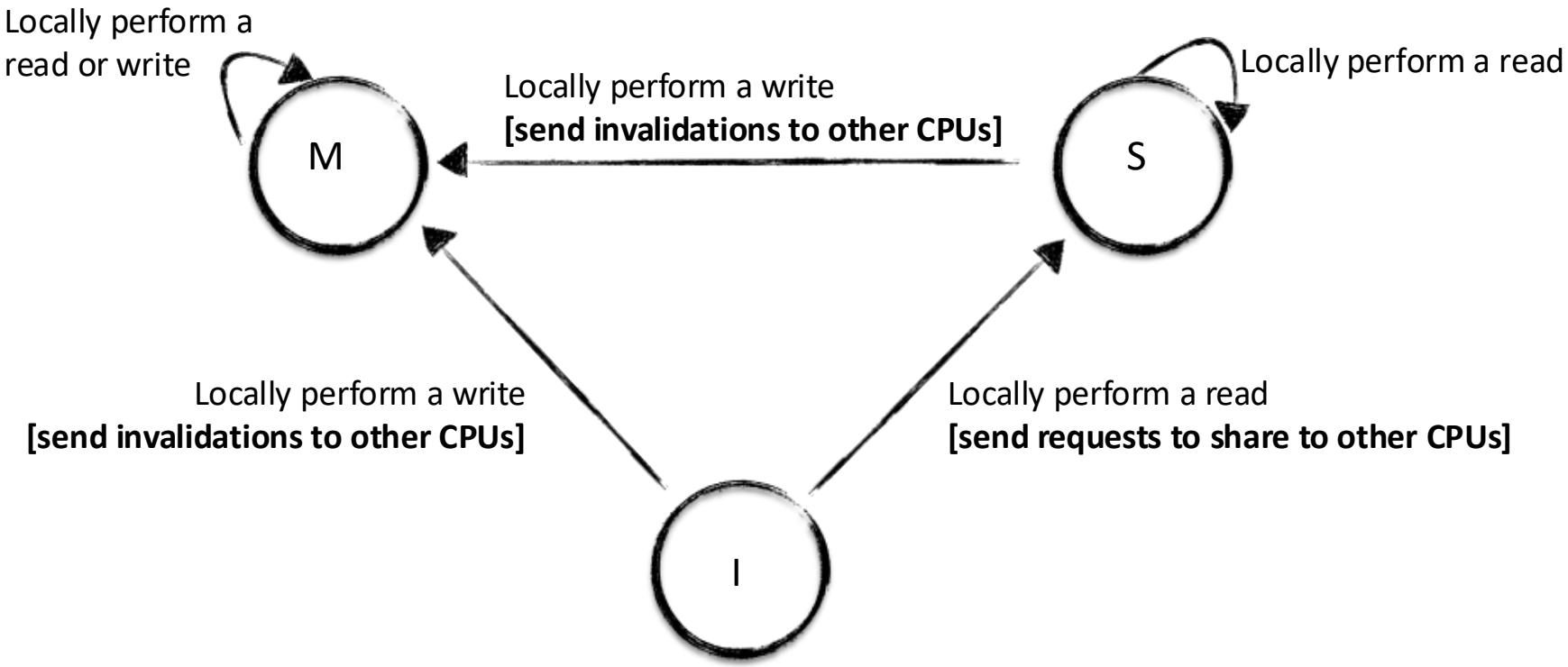
Per-line coherence states

Cache Coherence Protocol



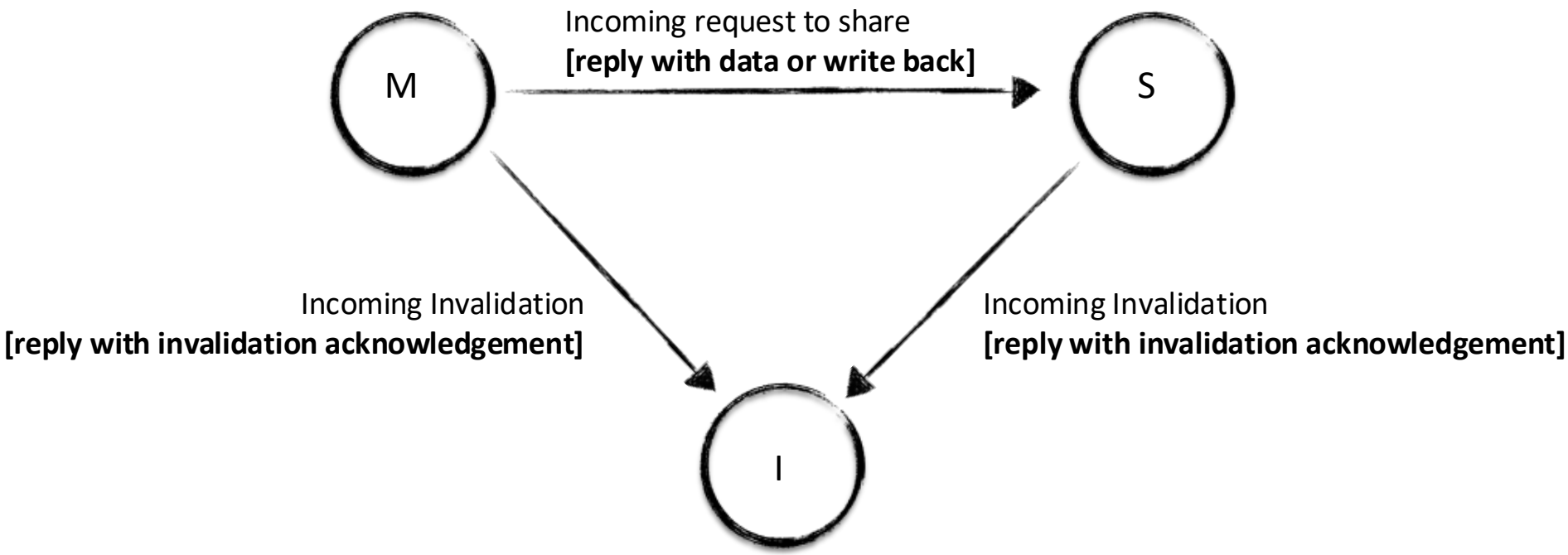
Cache Coherence Protocol

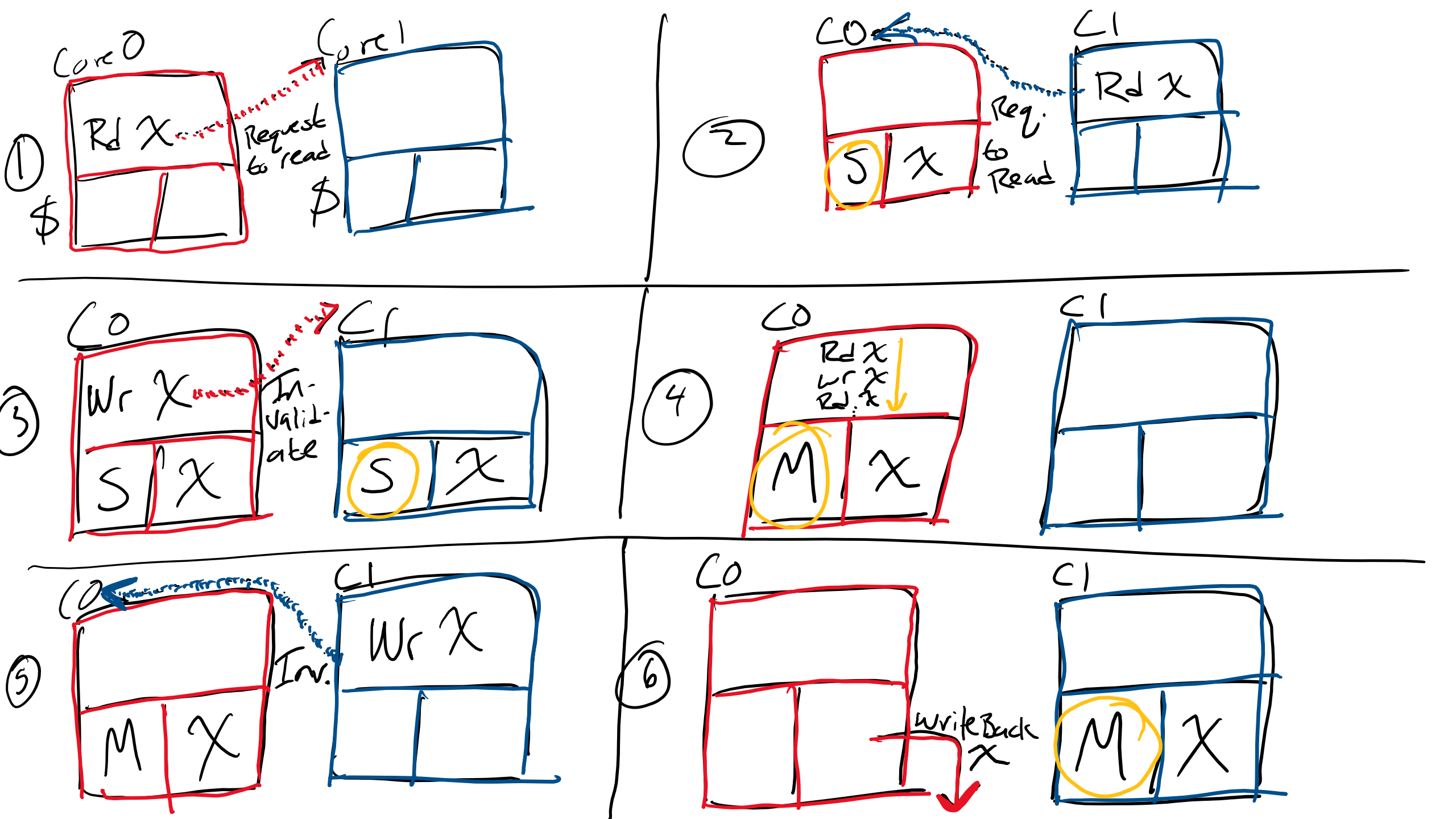
Local operations perspective



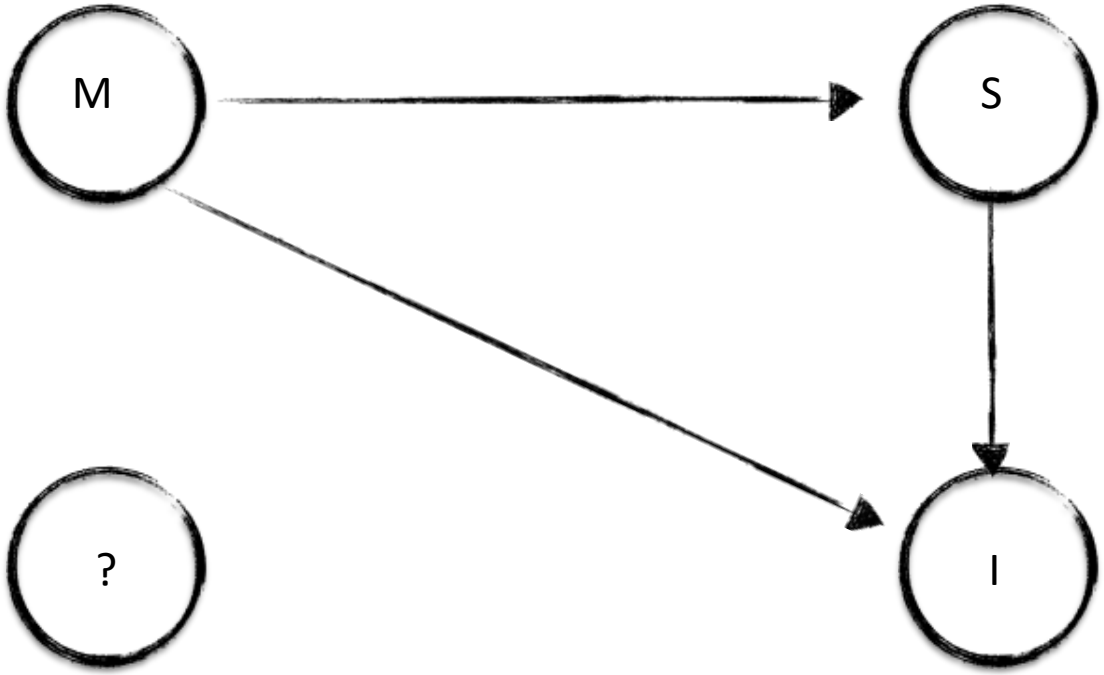
Cache Coherence Protocol

Remote operations perspective

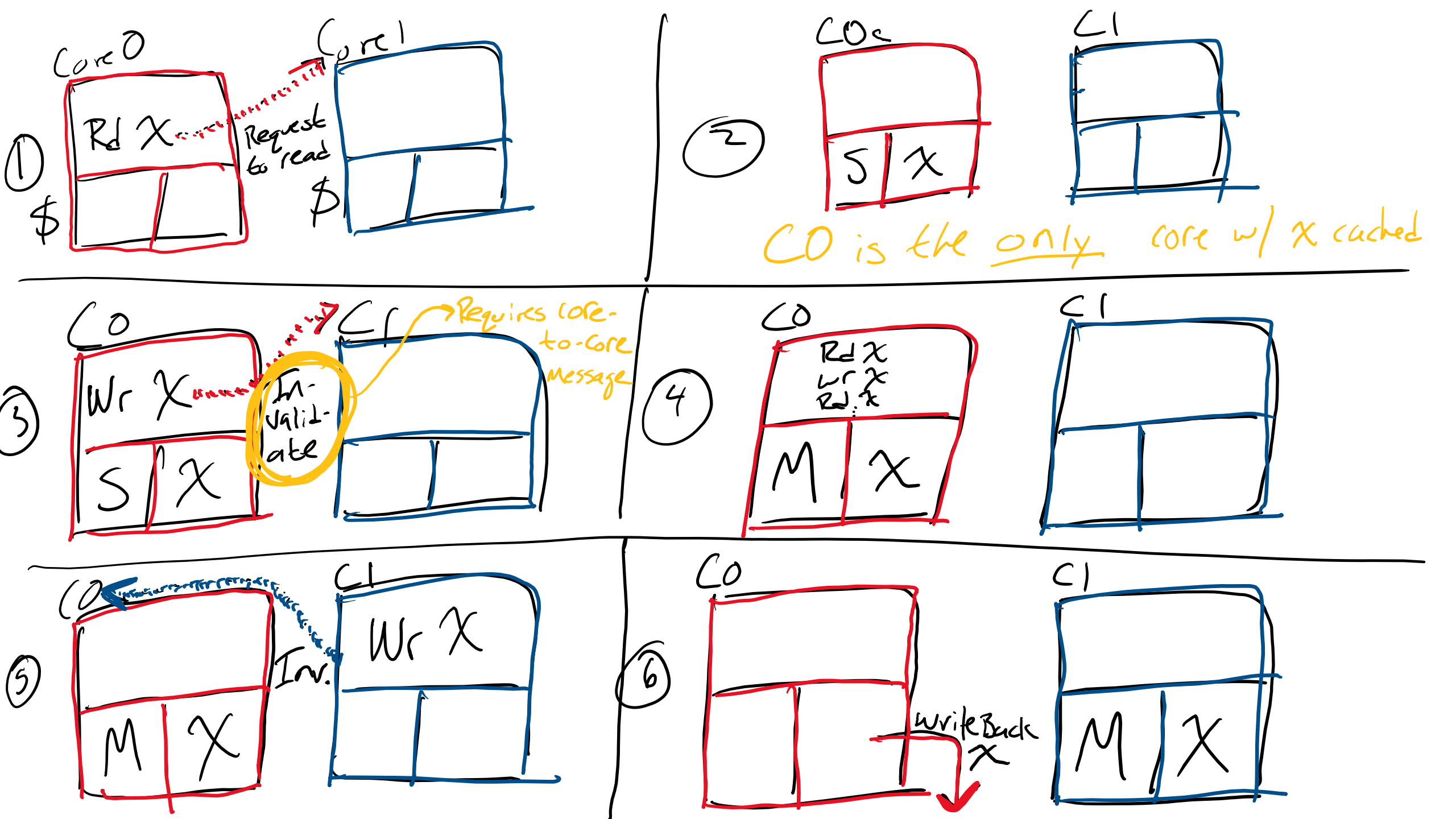



Core 0
Core 1
Rd X
Request to read
$
$
1
C0
C1
Rd X
Req. to Read
S
X
2
C0
C1
Wr X
In-valid-ate
S
X
3
C0
C1
Rd X
Wr X
Rd X
M
X
4
C0
C1
Wr X
Inv.
M
X
5
C0
C1
WriteBack X
M
X
6

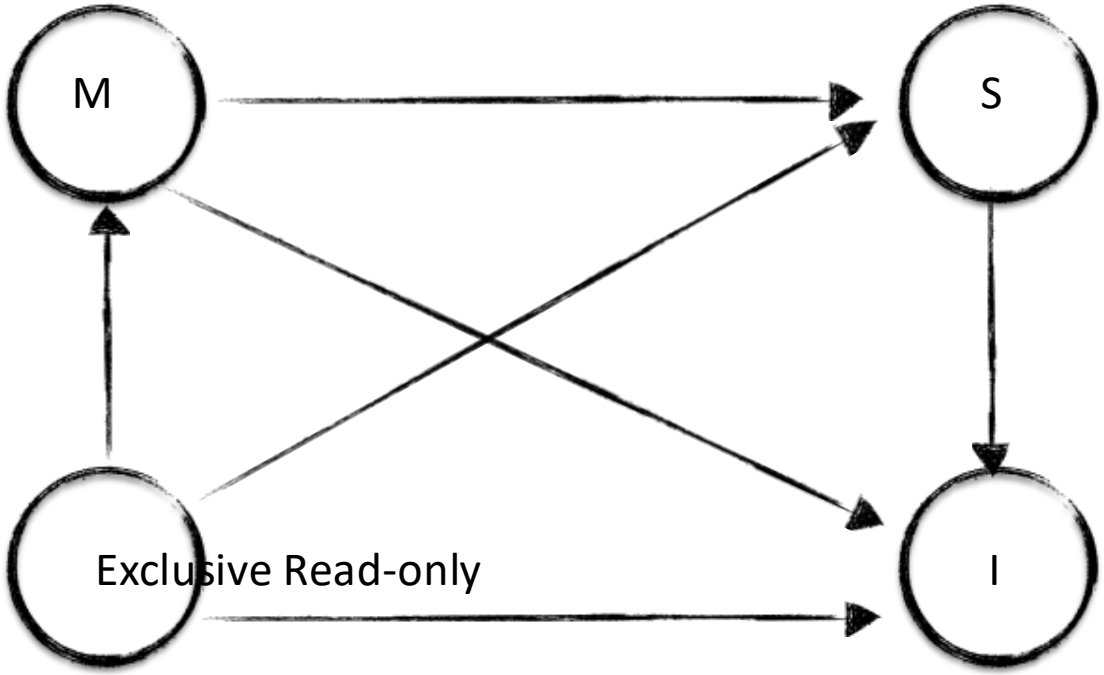
Can we design another state?



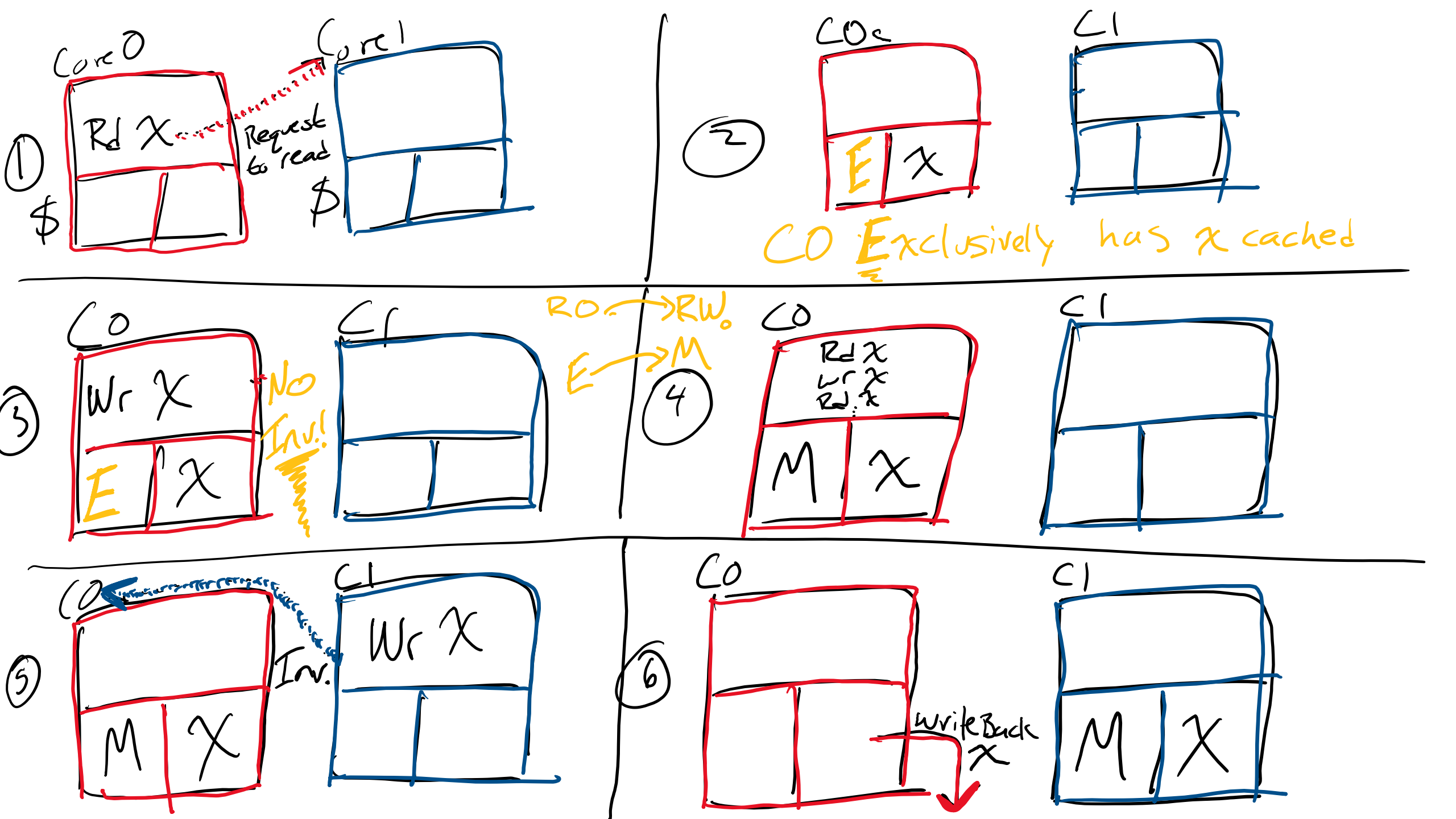
What should we optimize?


Core 0
Core 1
Rd X
Request to read
$
$
1
2
C0
C1
S
X
C0 is the only core w/ X cached
3
C0
C1
Wr X
In-valid-ate
Requires core-to-core message
S
X
4
C0
C1
Rd X
Wr X
Rd X
M
X
5
C0
C1
Inv.
Wr X
M
X
6
C0
C1
WriteBack X
M
X

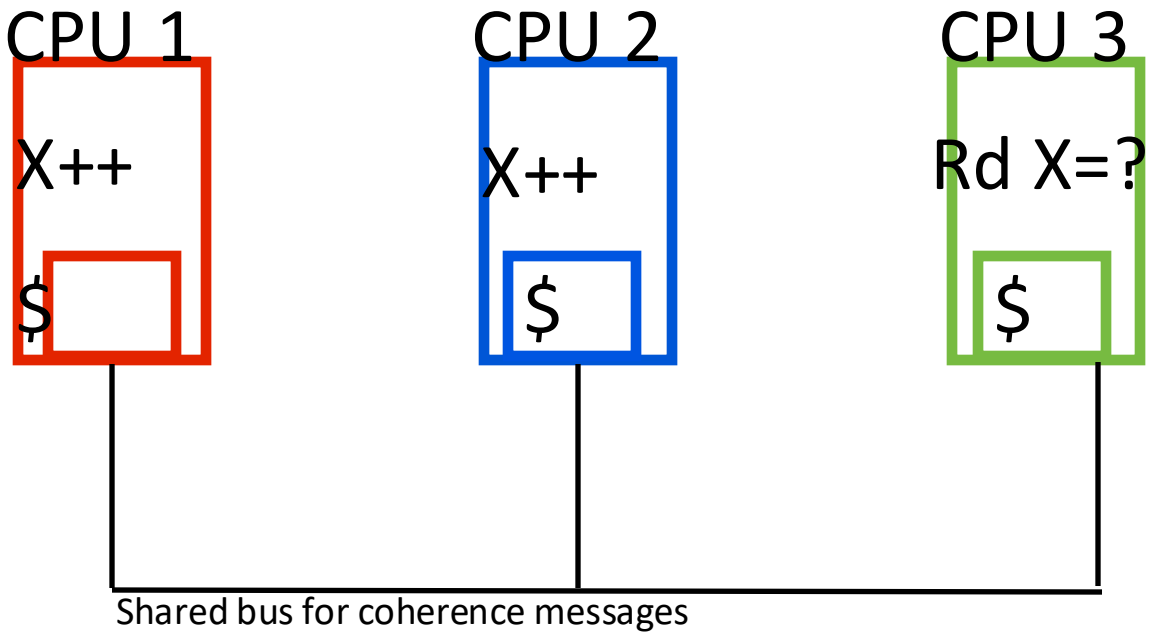
Exclusive read-only avoids invalidation messages



(Benefit: no invalidation required
to transition from E->M, like from S->M)

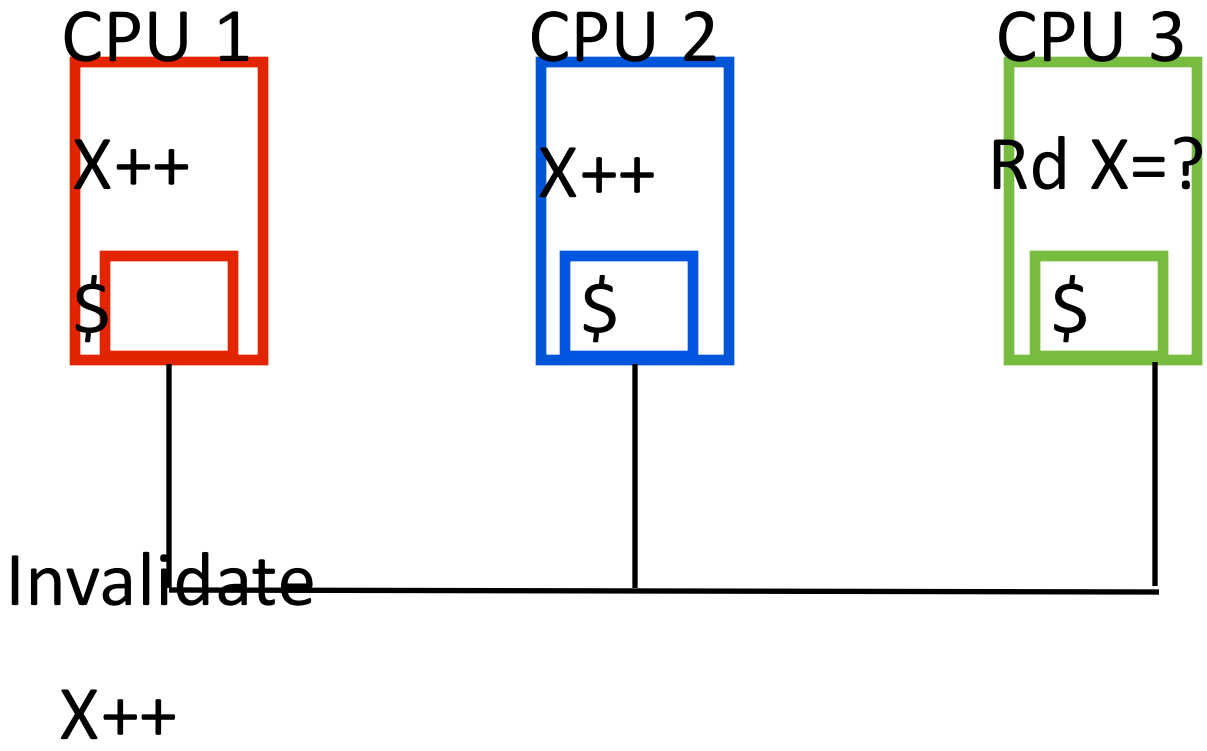

Core 0
Core 1
Rd X
Request to read
$
$
1
2
C0
C1
E
X
C0 Exclusively has x cached
3
C0
C1
Wr X
E
X
*No Inv!
RO → RW
E → M
4
C0
Rd X
Wr X
Rd X
M
X
C1
5
C0
C1
Wr X
Inv.
M
X
6
C0
C1
WriteBack X
M
X

Implementing the Protocol

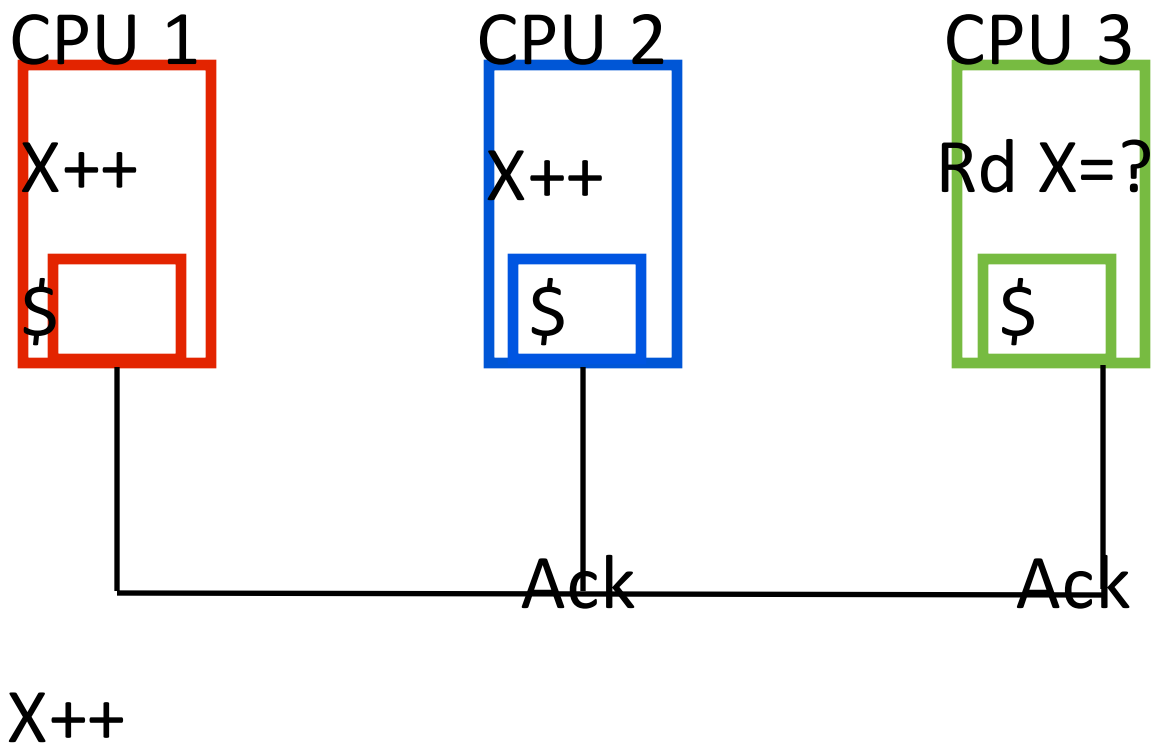


Snoopy Coherence

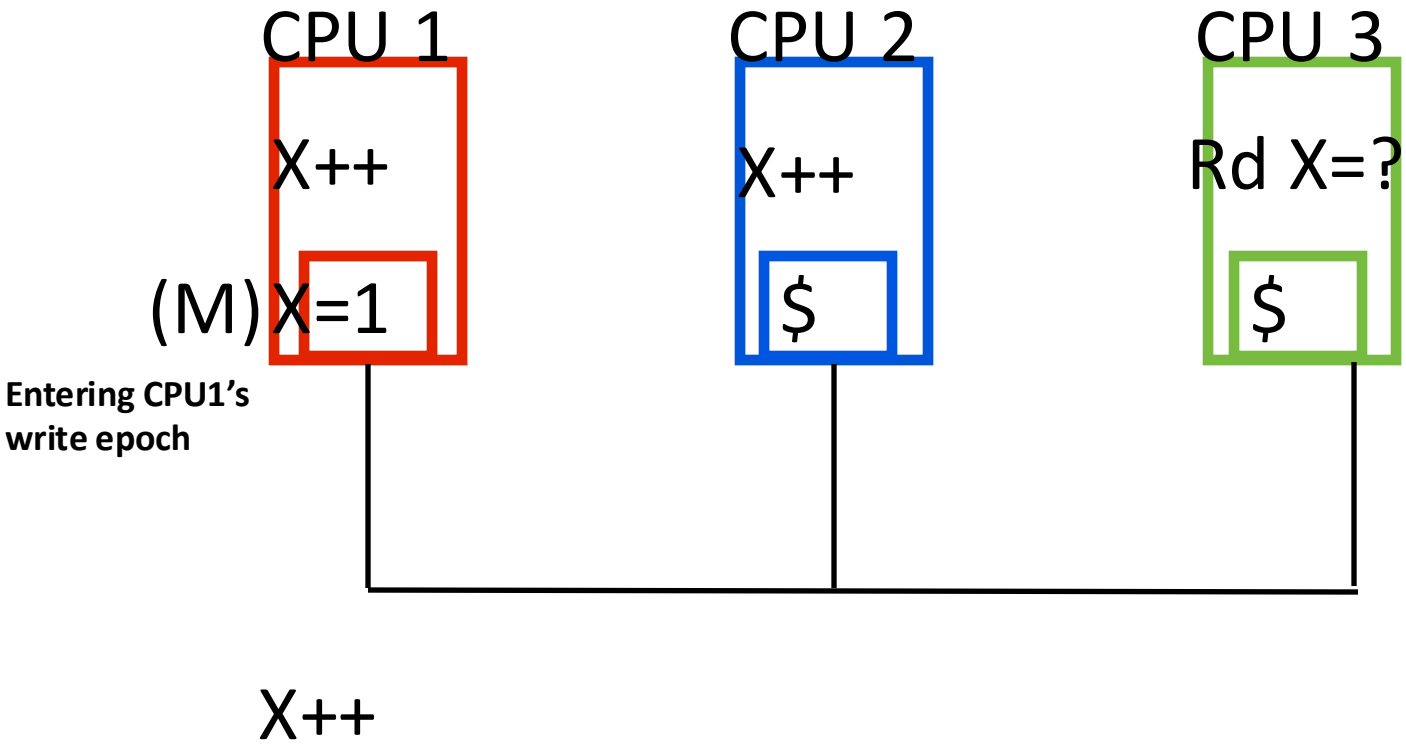
Implementing the Protocol



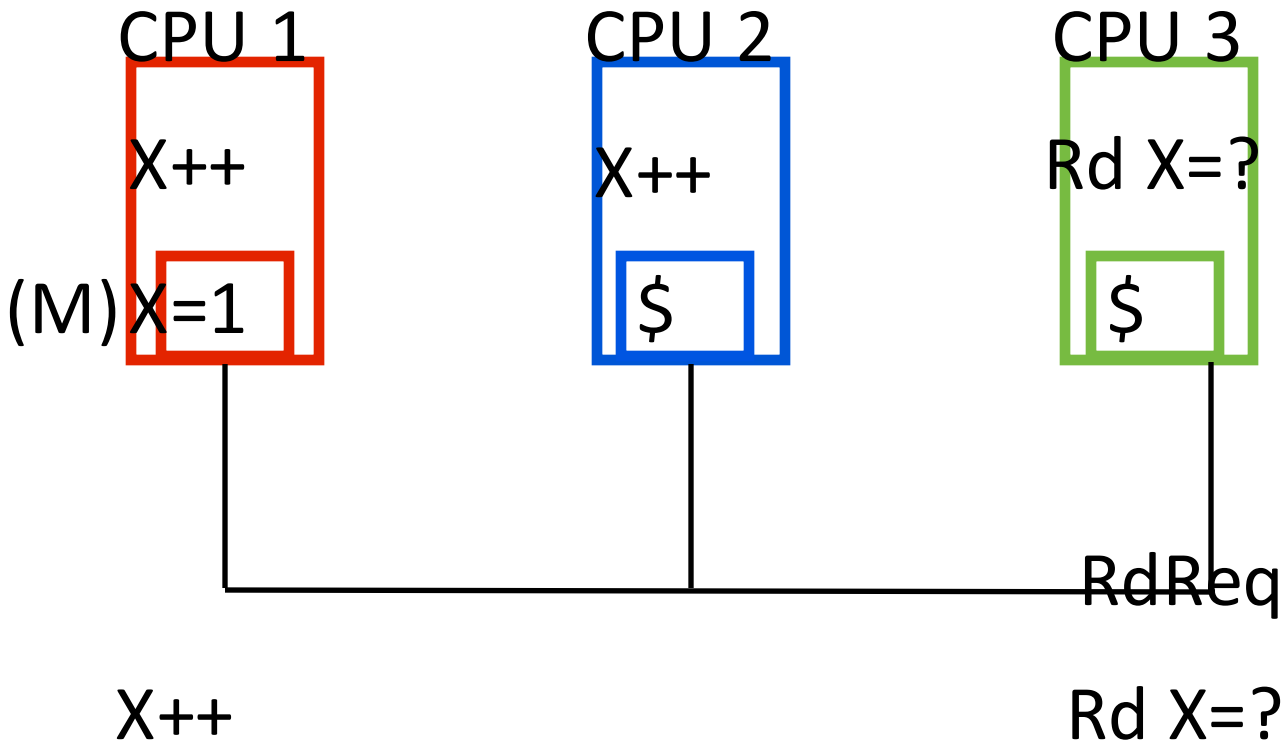
Implementing the Protocol



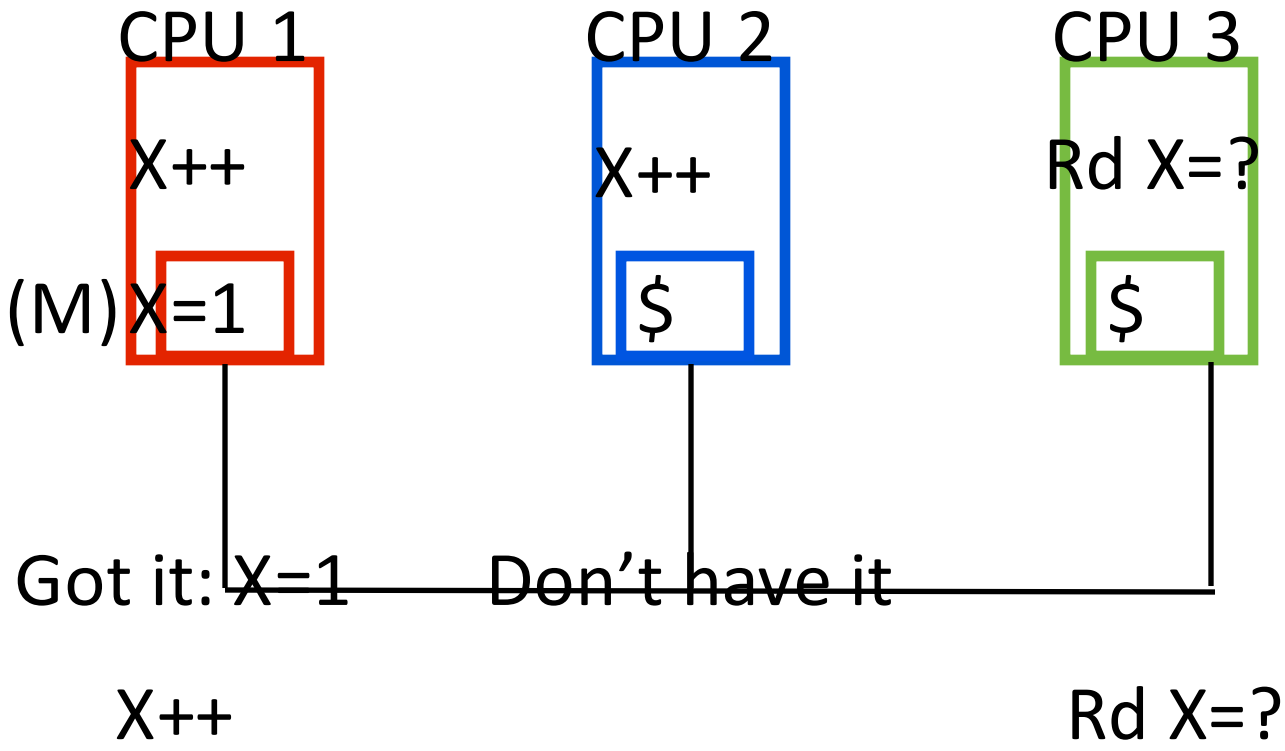
Implementing the Protocol



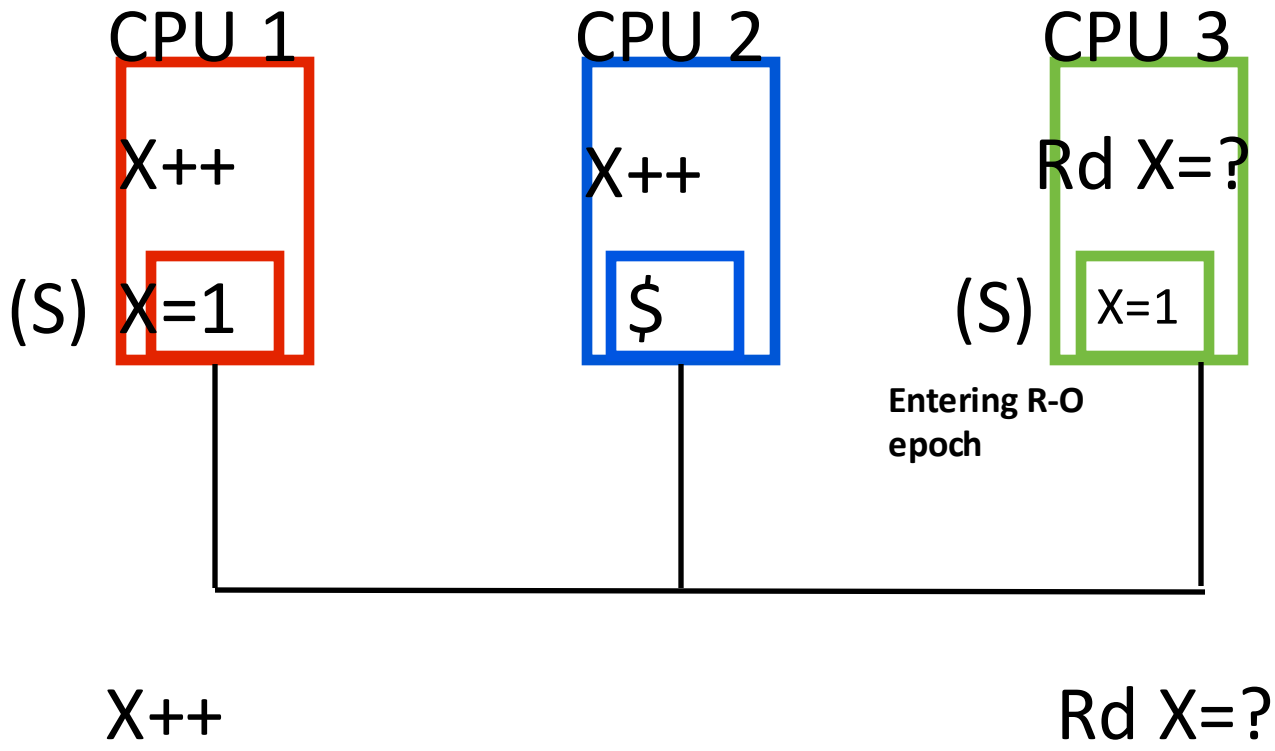
Implementing the Protocol



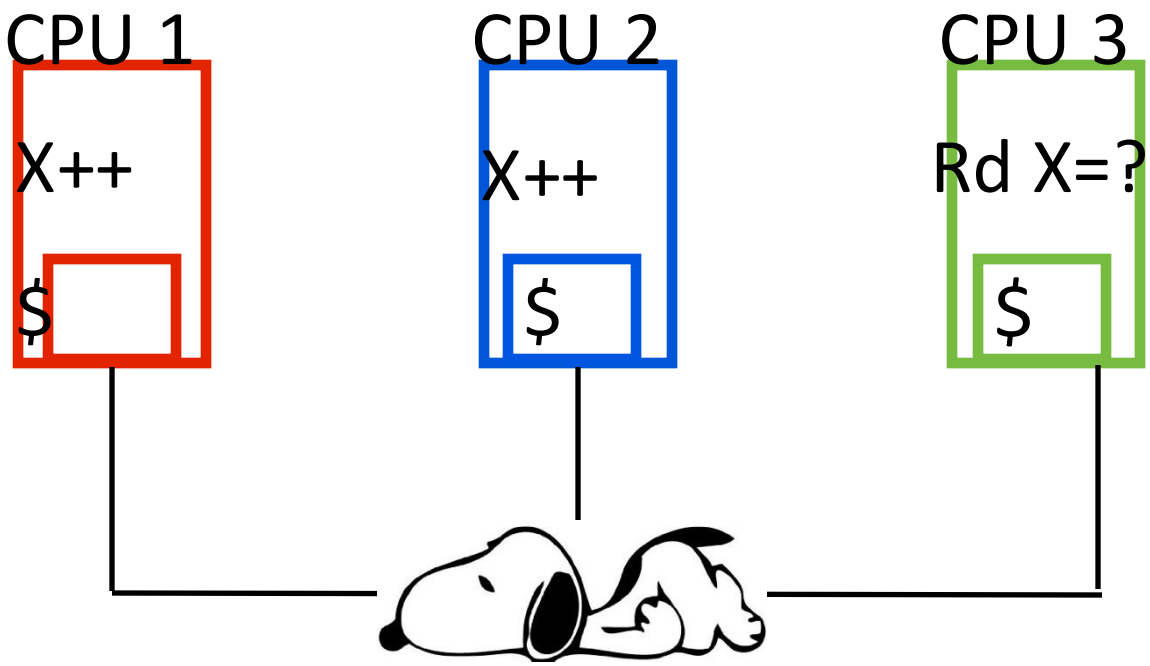
Implementing the Protocol



Implementing the Protocol

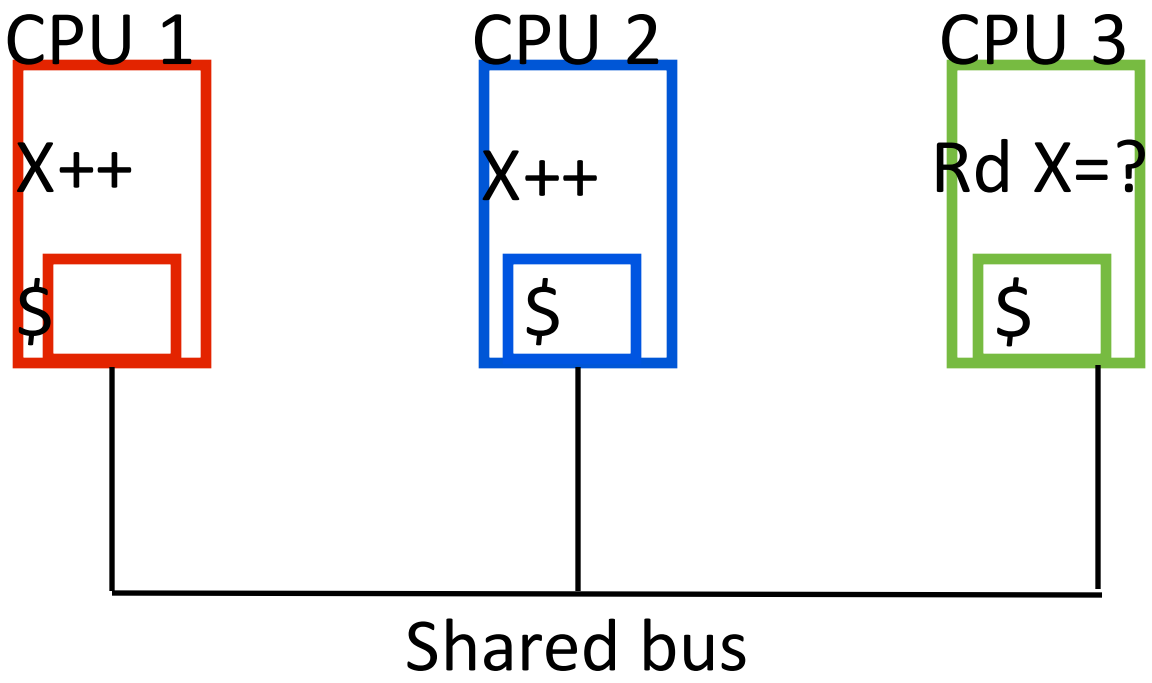


Implementing the Protocol



What sucks about Snoopy?

Implementing the Protocol



Bus limits scalability due to congestion and complex message arbitration

Figure 1-1. Uncore Sub-system Block Diagram of Intel Xeon Processor E5-2600 Family

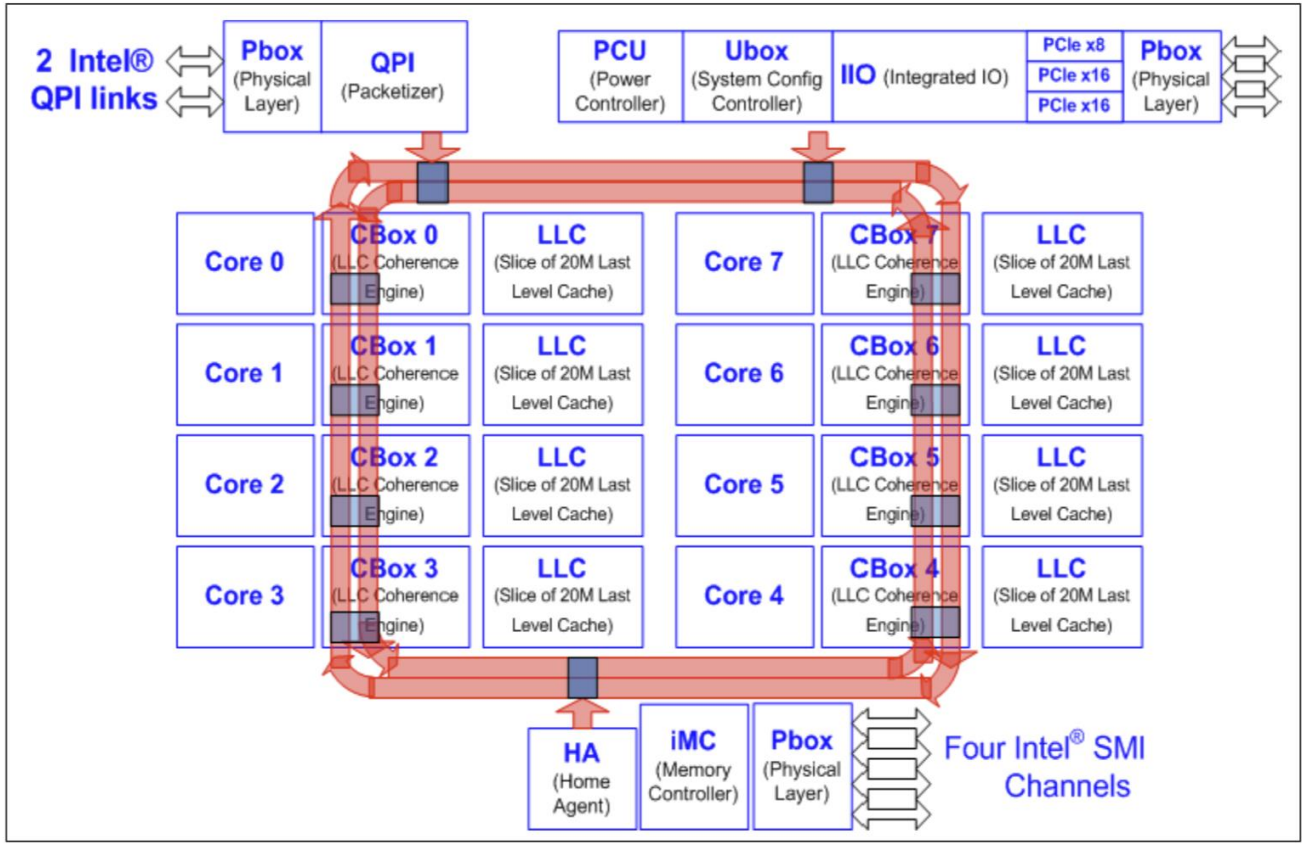
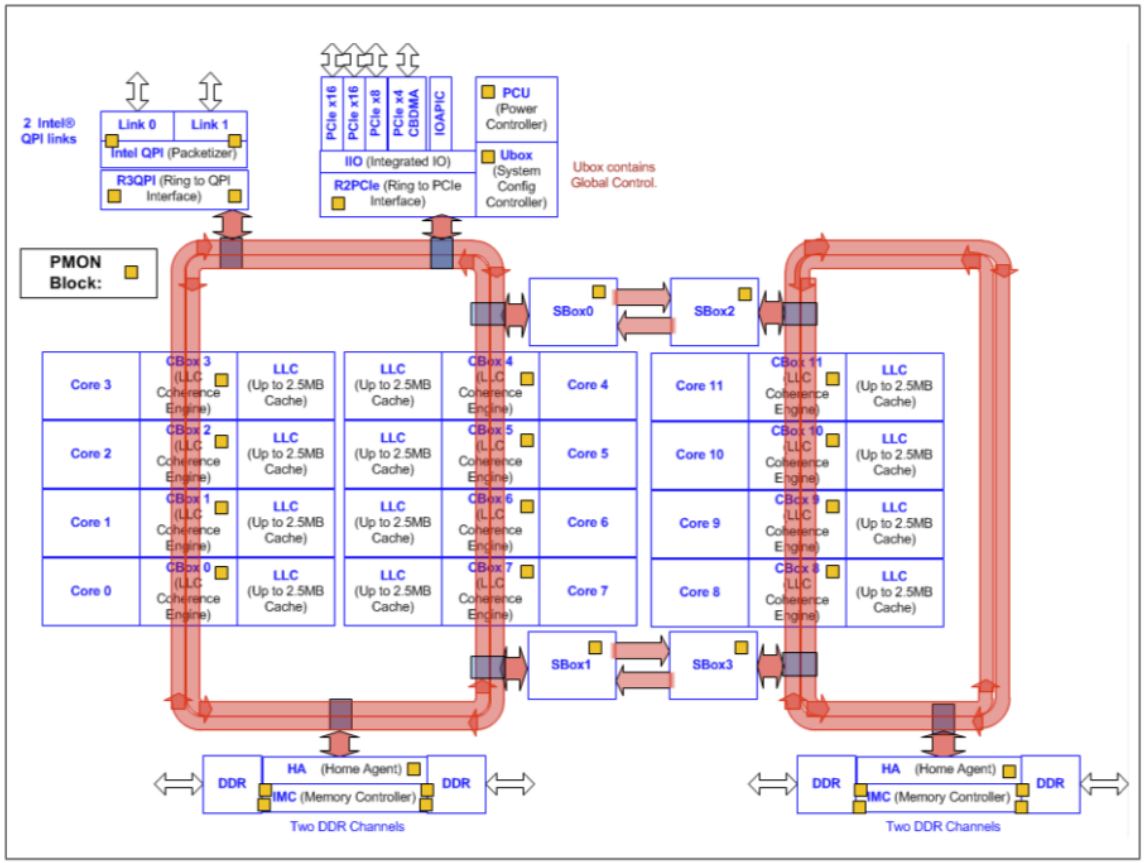
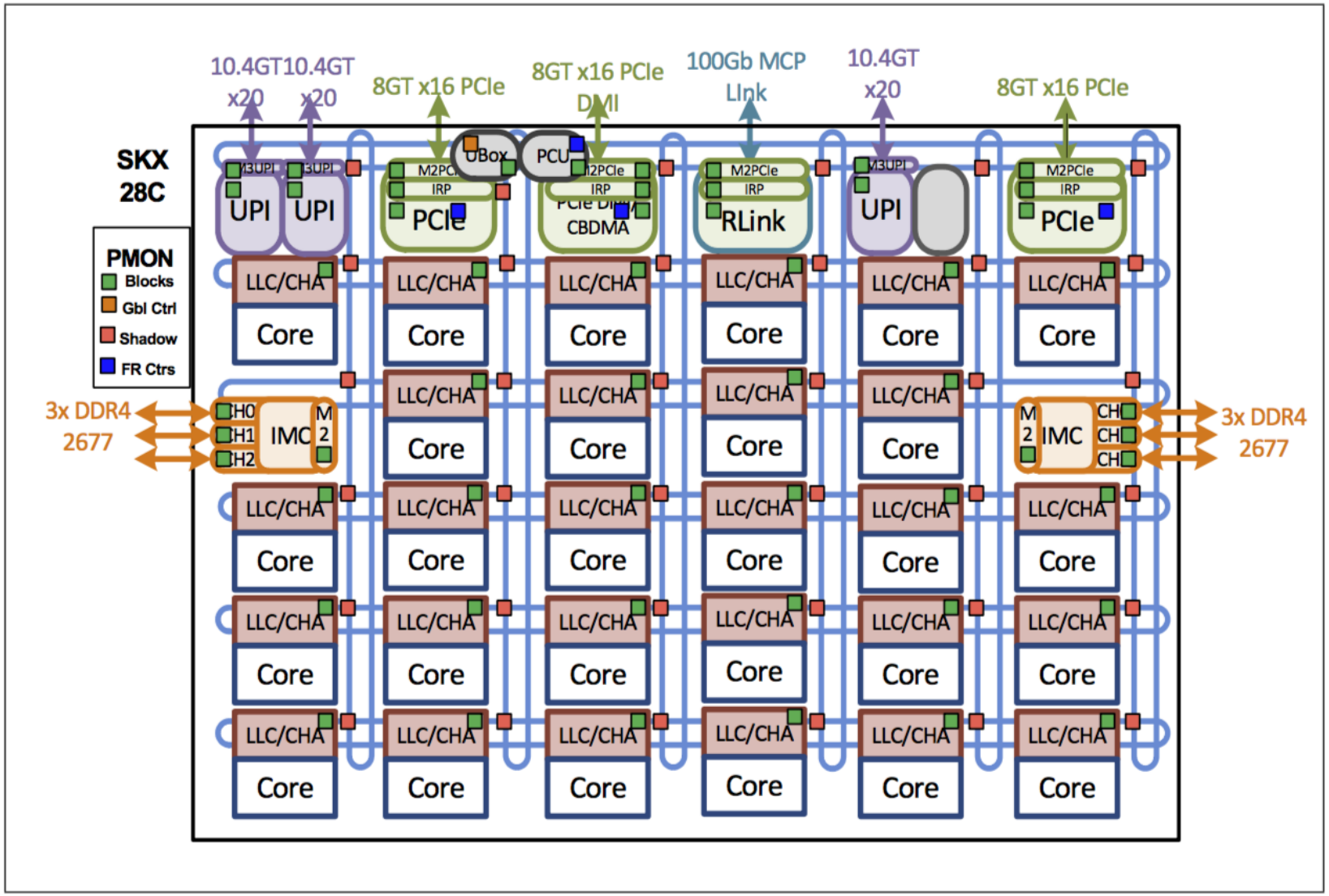


Figure 1-2. Intel® Xeon® Processor E5 v3-1600/2600/4600 Family -12C Block Diagram



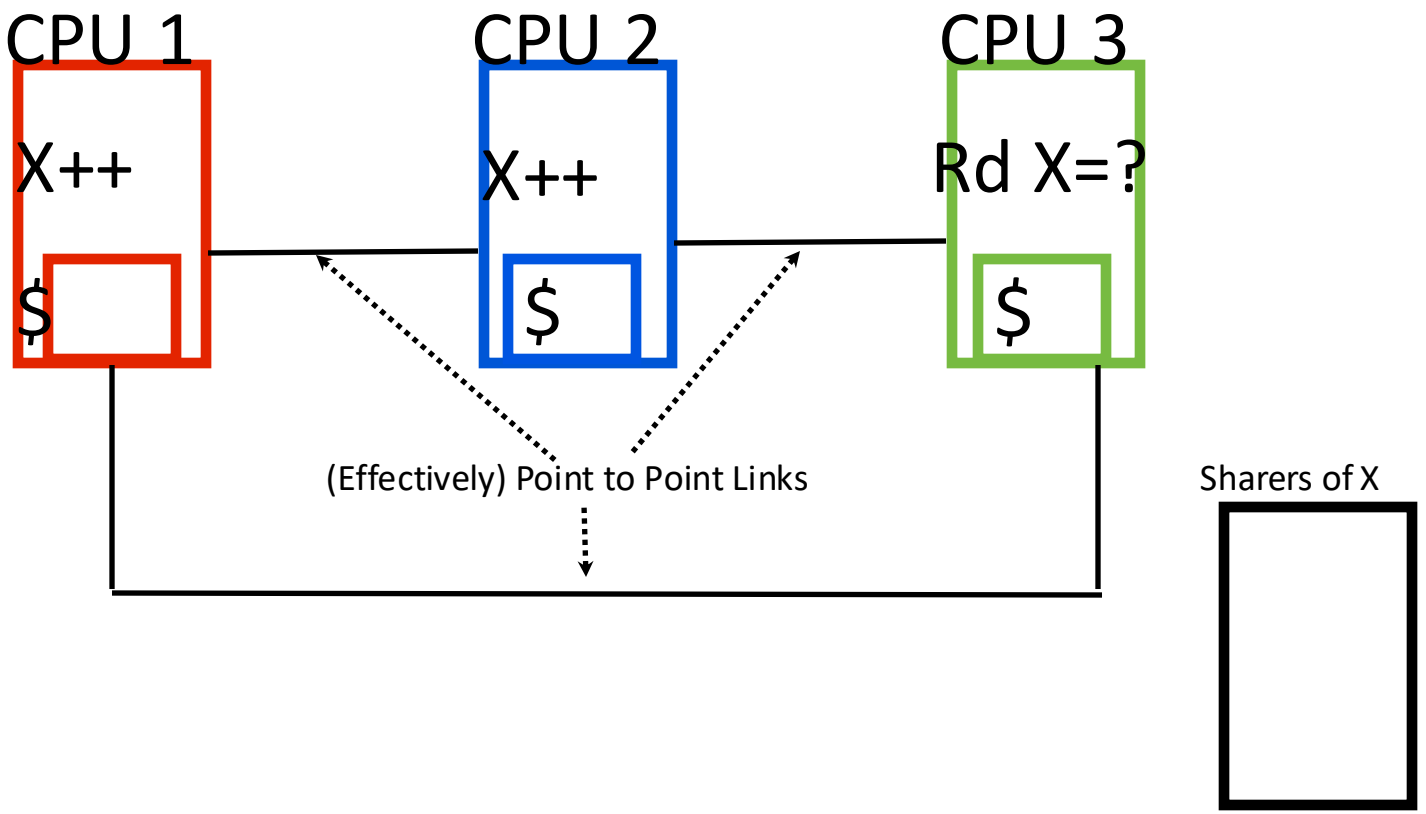
Intel Sandybridge Multiprocessor: bi-directional ring network

Figure 1-1. Intel® Xeon® Processor Scalable Memory Family - Block diagram for a 28C part



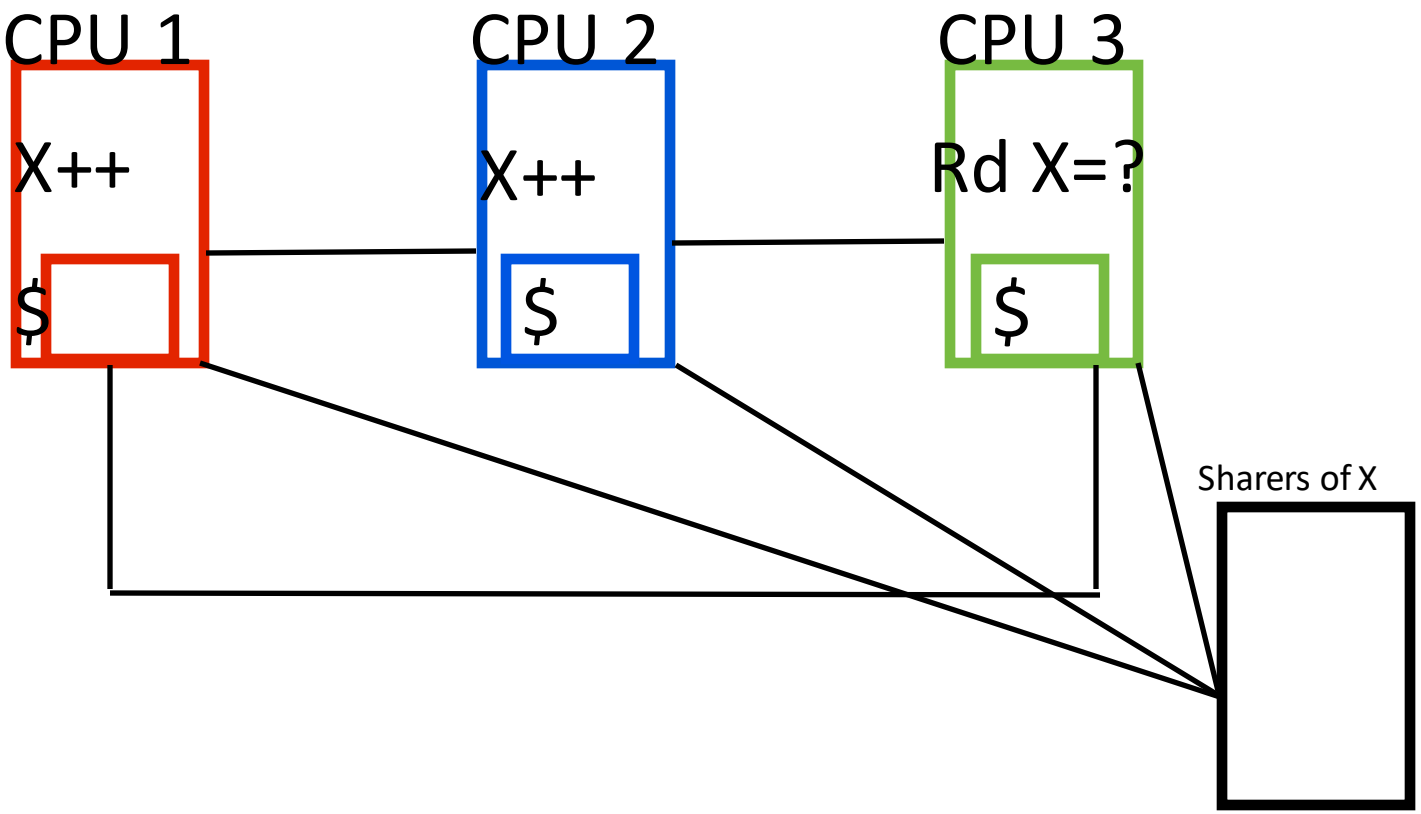
Skylake Xeon 2017 2D mesh

Implementing the Protocol



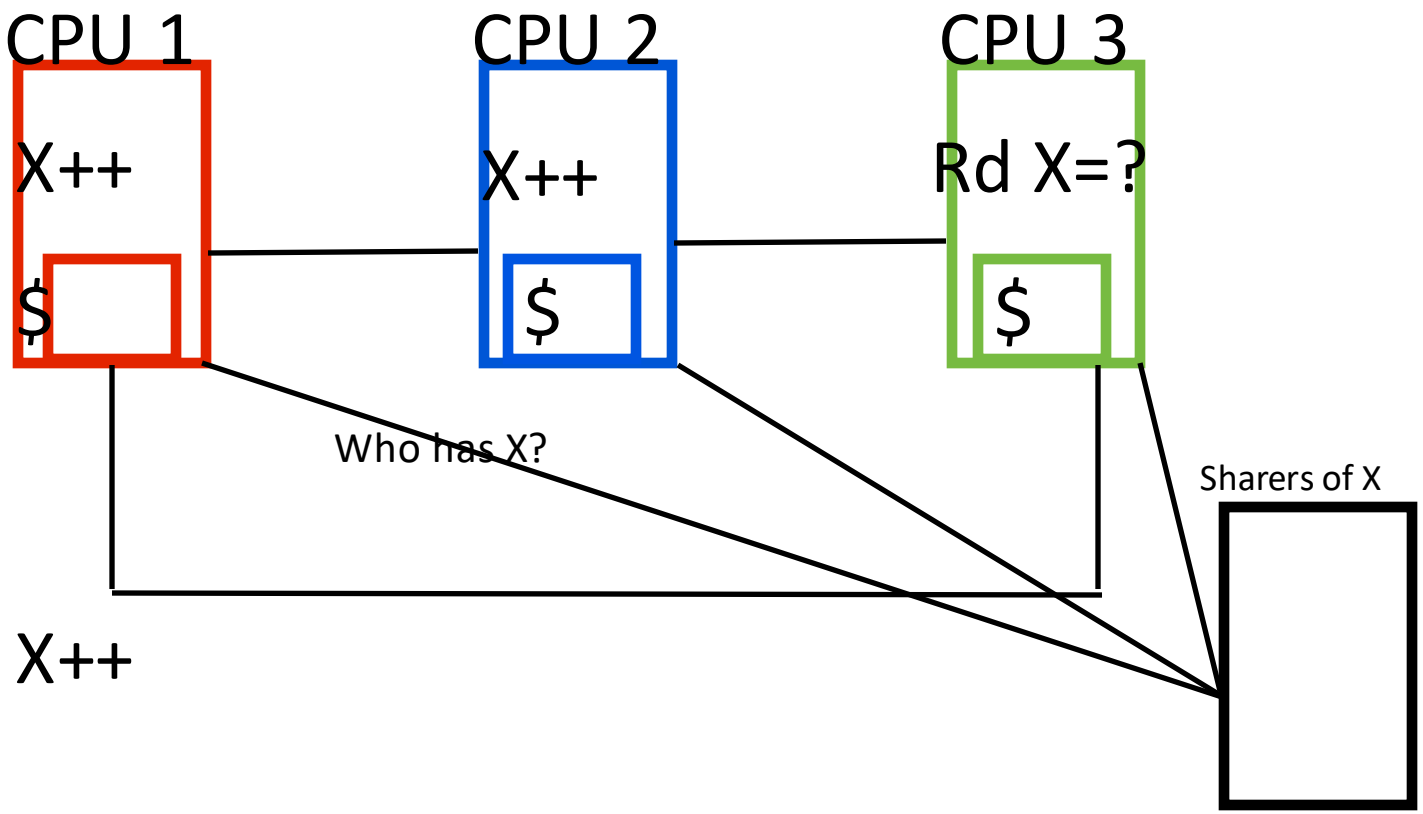
Directory-based

Implementing the Protocol



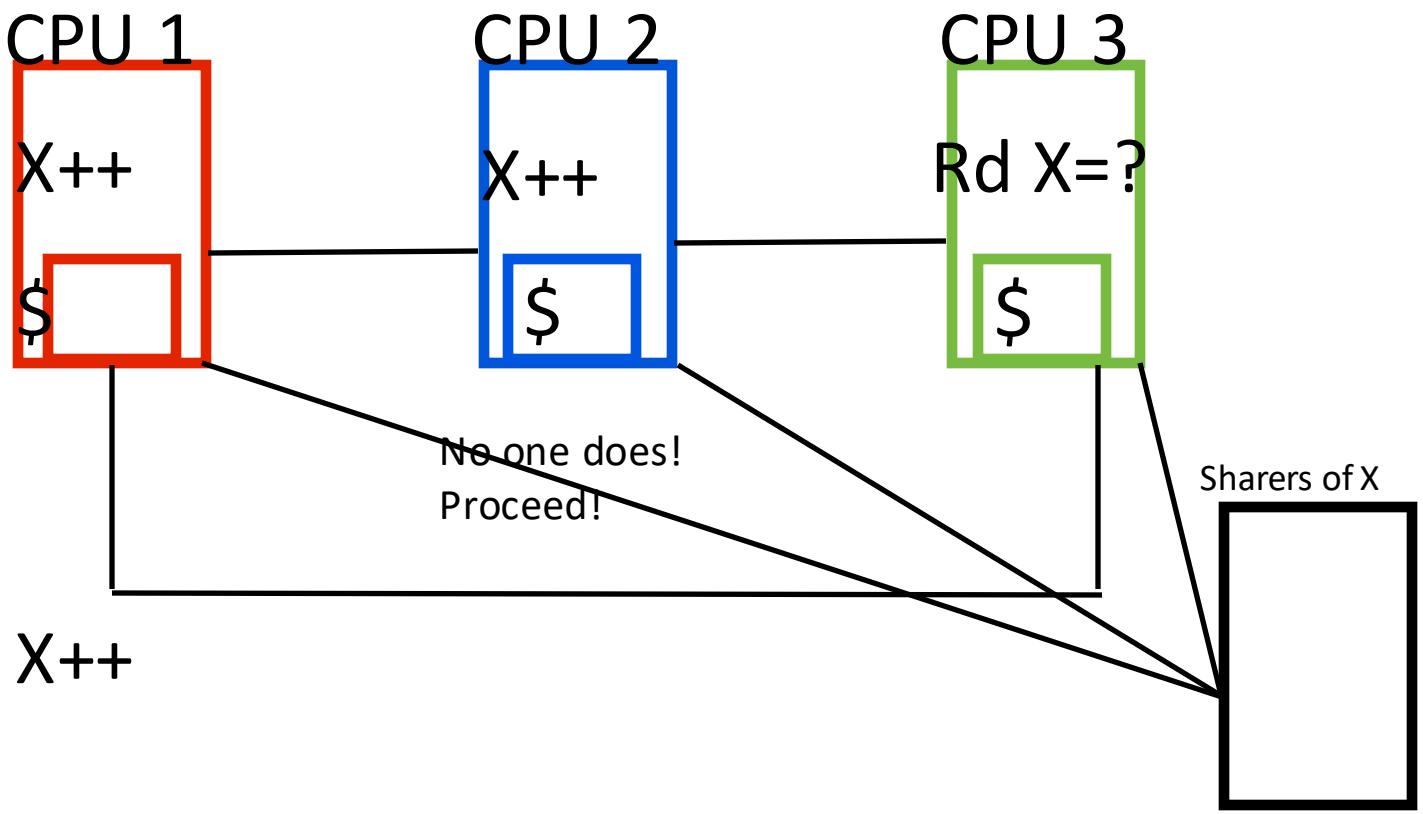
Directory-based

Implementing the Protocol



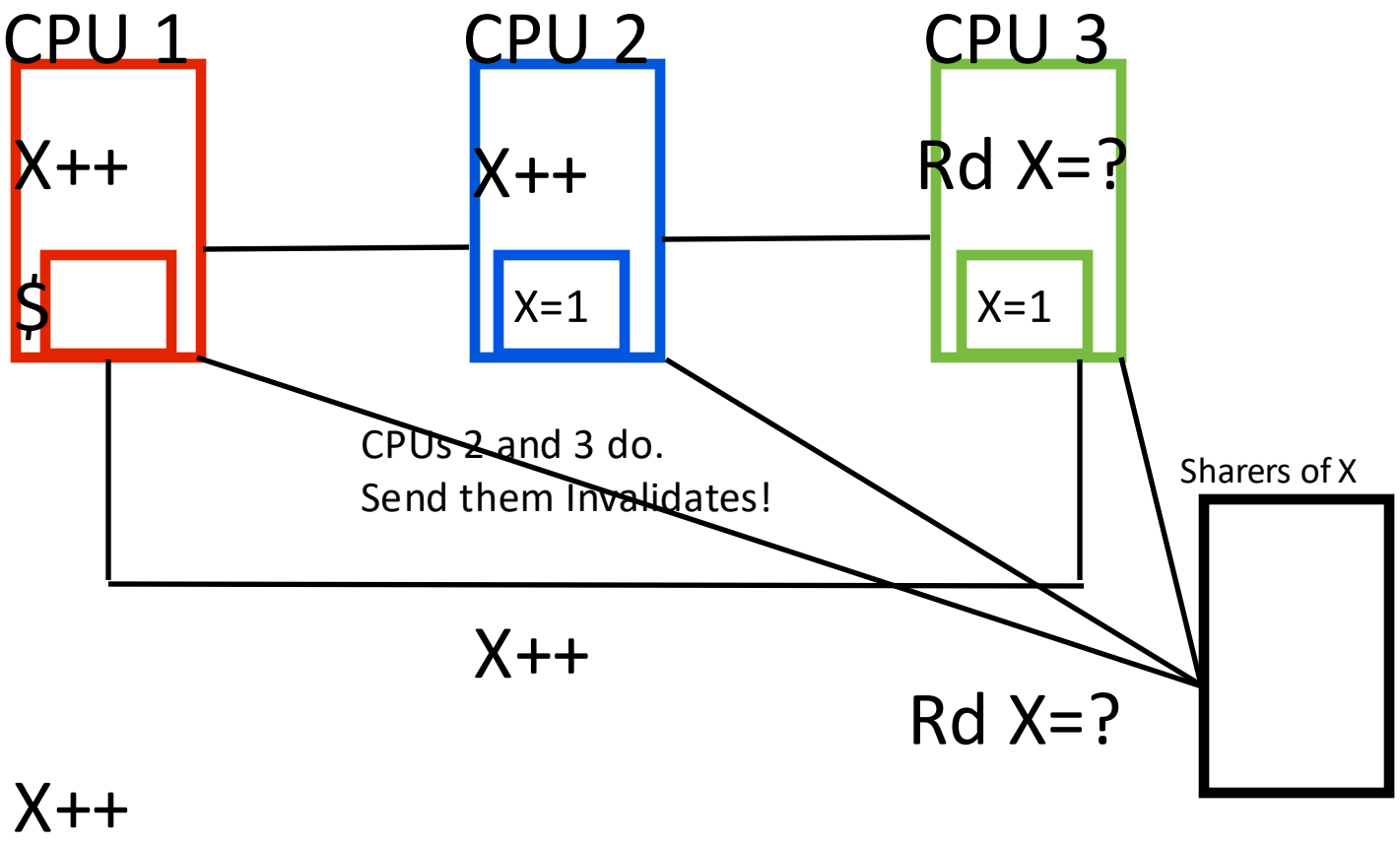
Directory-based

Implementing the Protocol



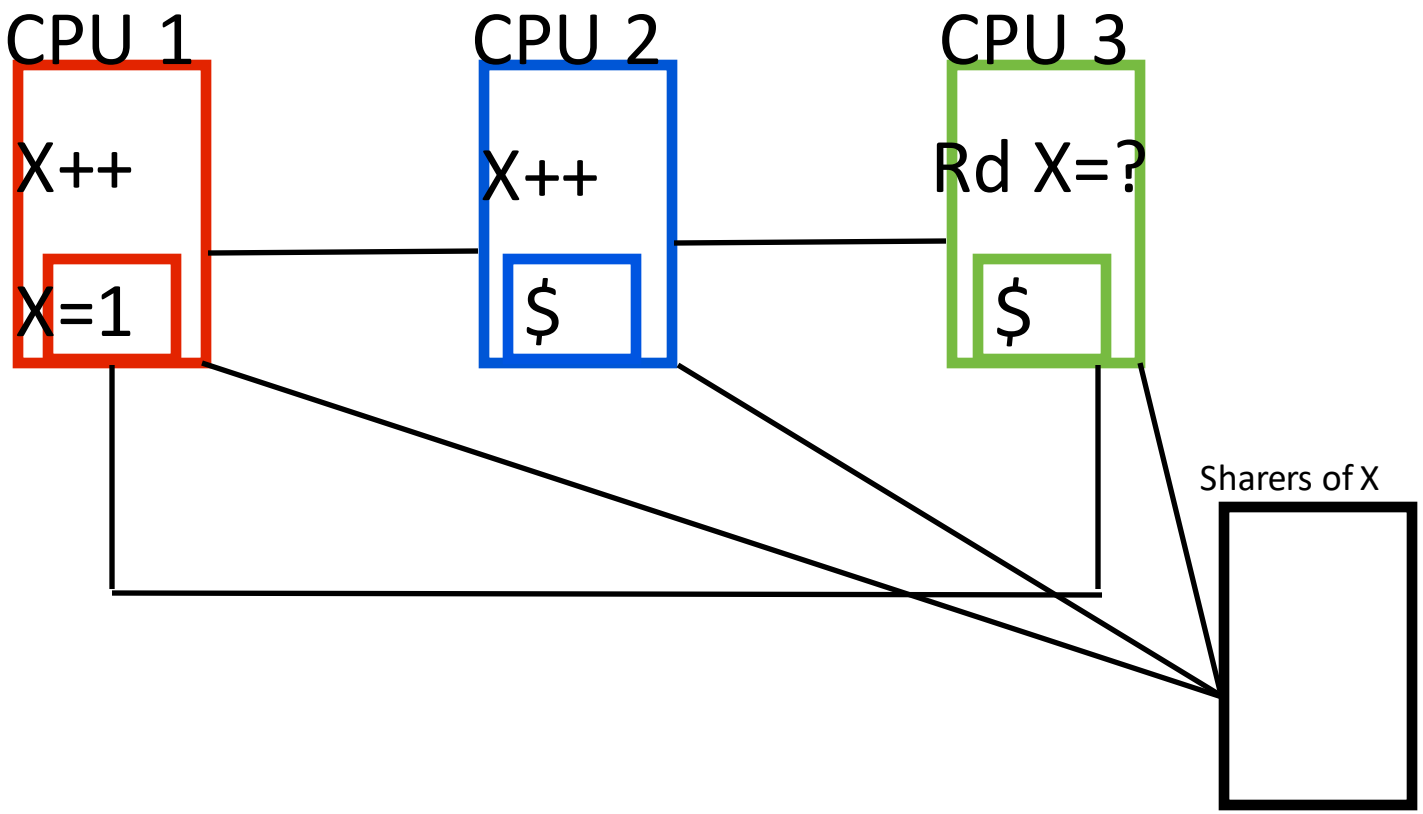
Directory-based

Implementing the Protocol



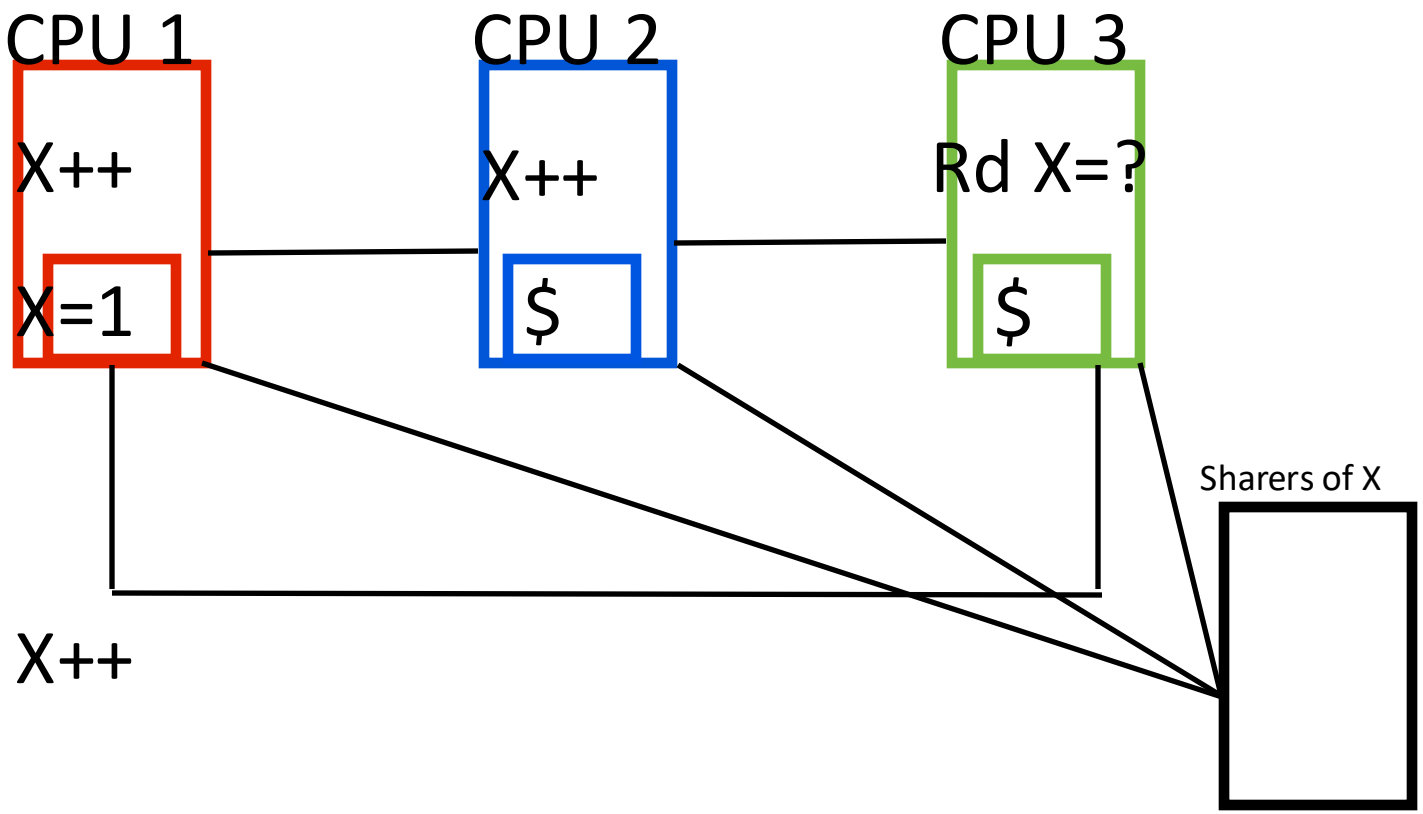
Directory-based

Implementing the Protocol



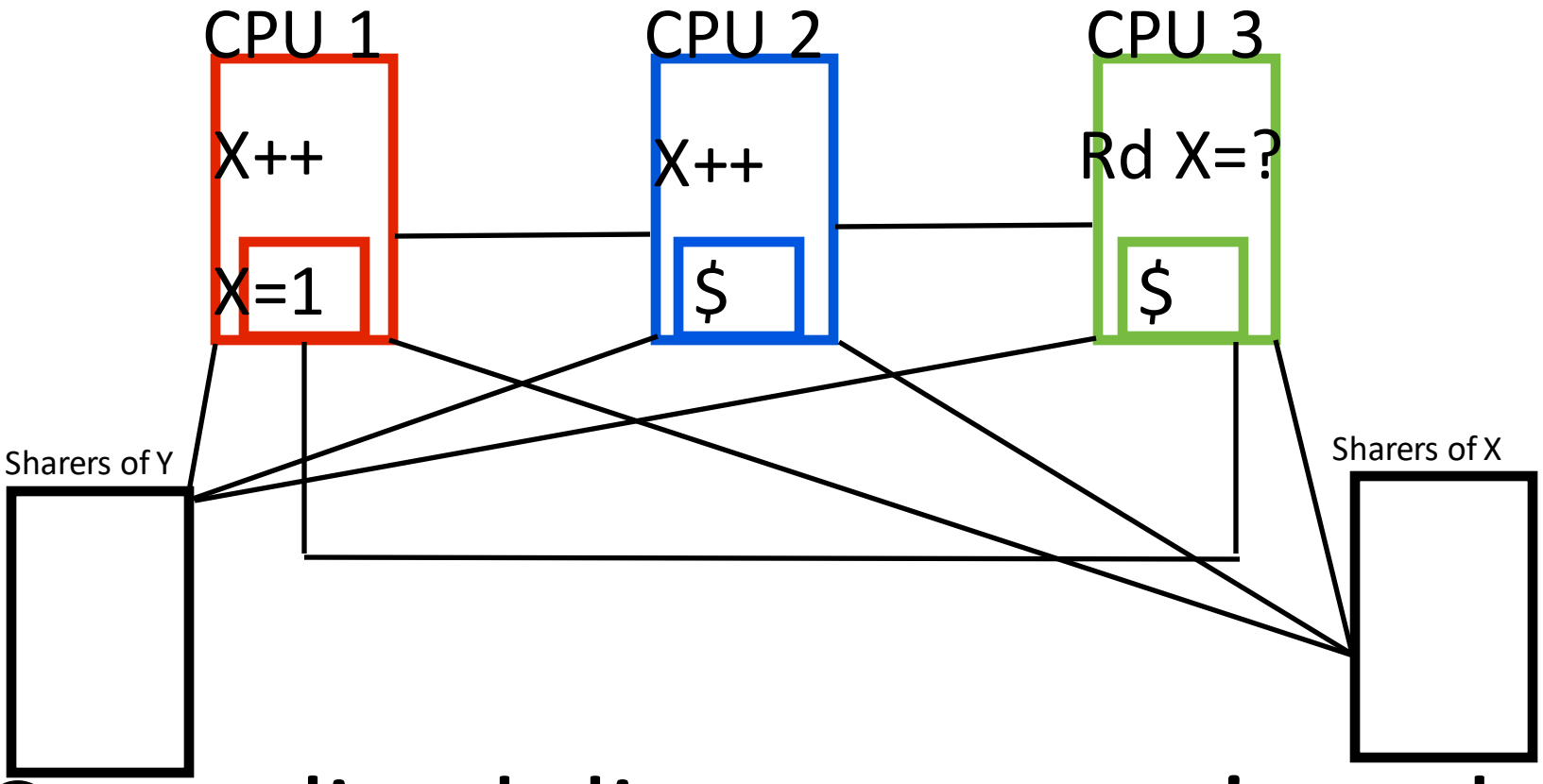
Benefit: No broadcast on shared bus

Implementing the Protocol



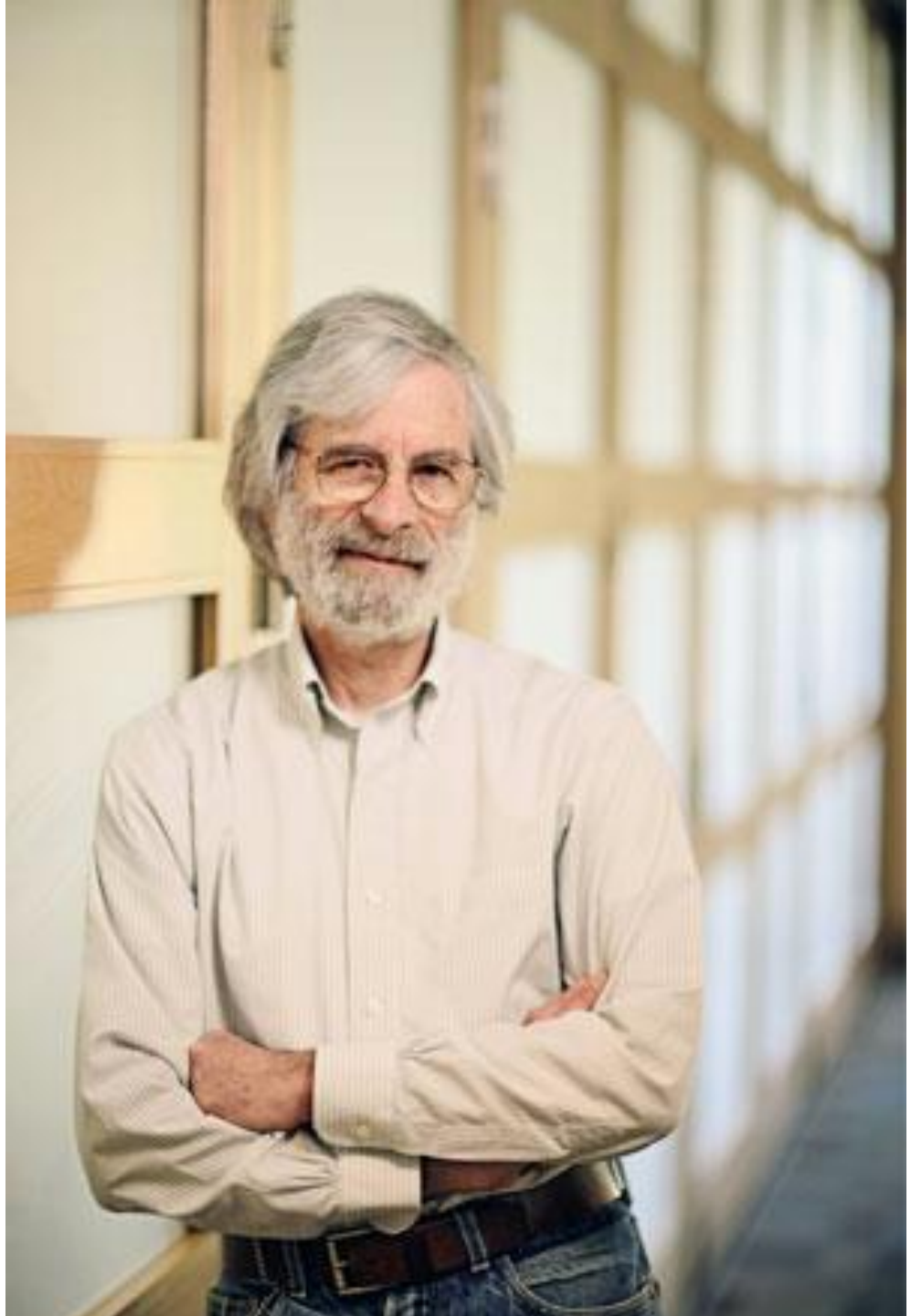
Drawbacks?

Implementing the Protocol



Centralized directory won't scale

(In Practice: Distribute Directory)



"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**."

Excerpt from "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program"
LESLIE LAMPORT, 1979

"The **memory consistency model** of a shared-memory system specifies the **order in which memory operations will appear to execute to the programmer.** The memory consistency model affects the process of writing parallel programs and forms an integral part of the entire system, including the architecture, the compiler, and the programming language."

Excerpt from "Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems"
Sarita Adve, et al, 1999

Memory Consistency

Memory Consistency Model

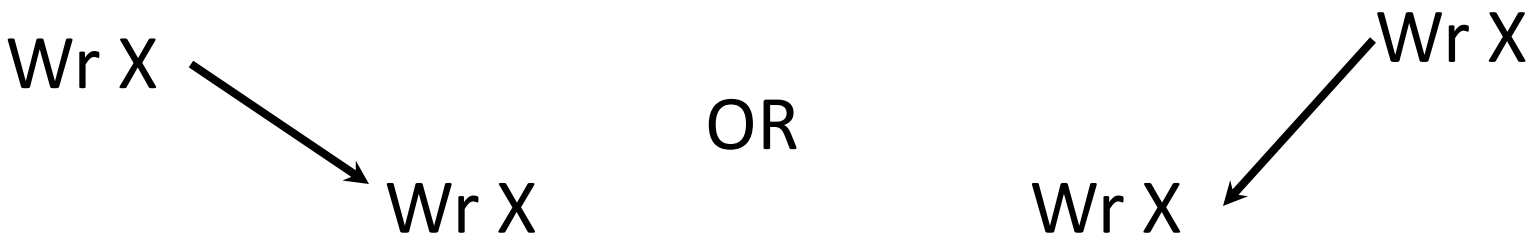
Informal Definitions:

> “Defines the value a read operation may read at each point during the execution”

> “Defines the set of legal observable orders of memory operations during an execution”

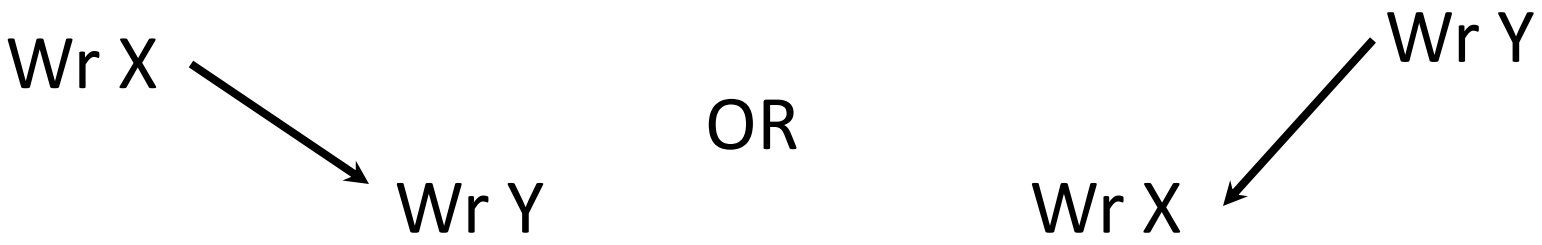
> “Defines which reorderings of memory operations are permitted”

Coherence is Ordering



Coherence defines the set of legal orders of accesses to a single memory location

Consistency is Ordering



Consistency defines the set of legal orders of accesses to multiple memory locations

Sequential Consistency (SC)

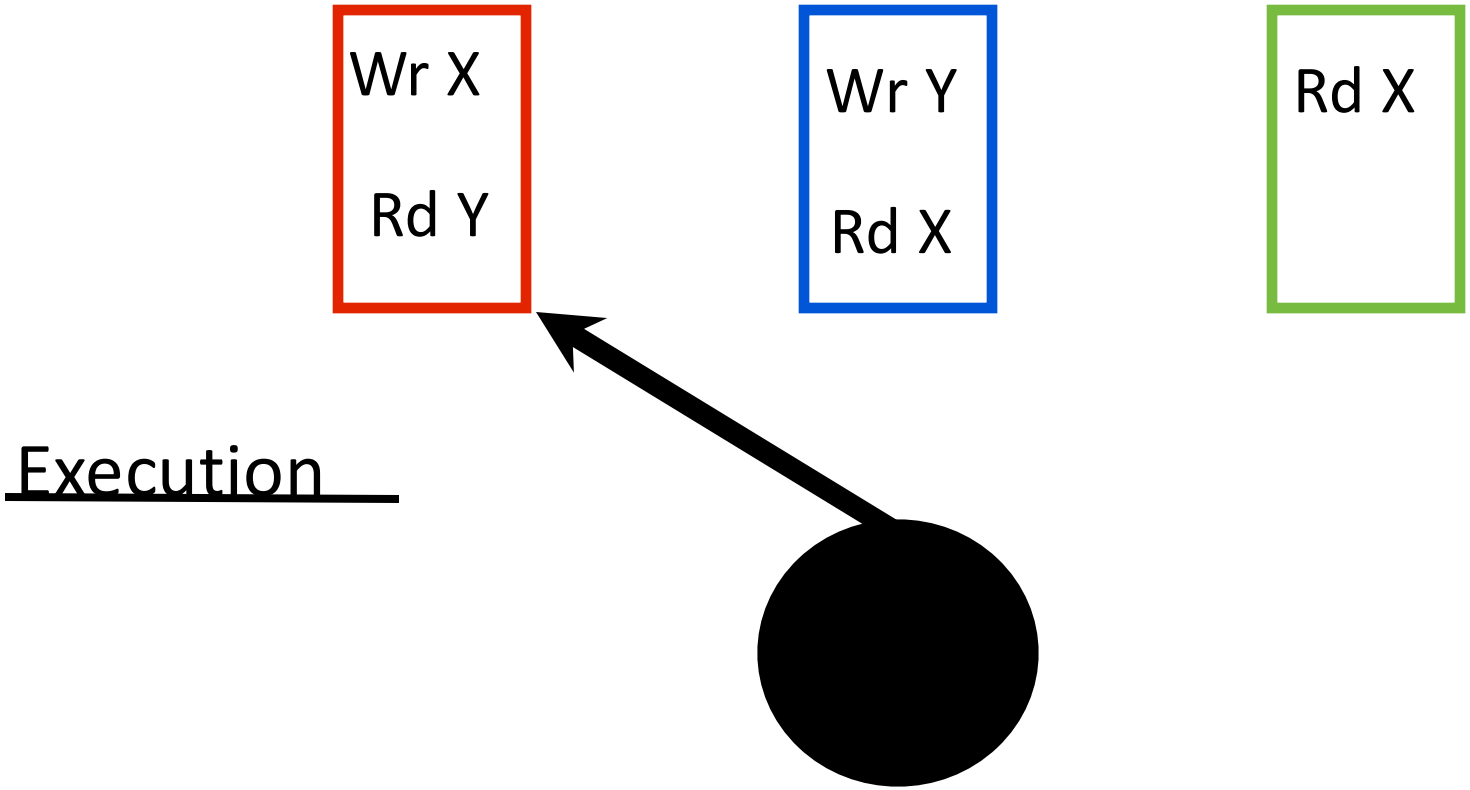
The simplest, most intuitive memory consistency model

Two Invariants to SC:

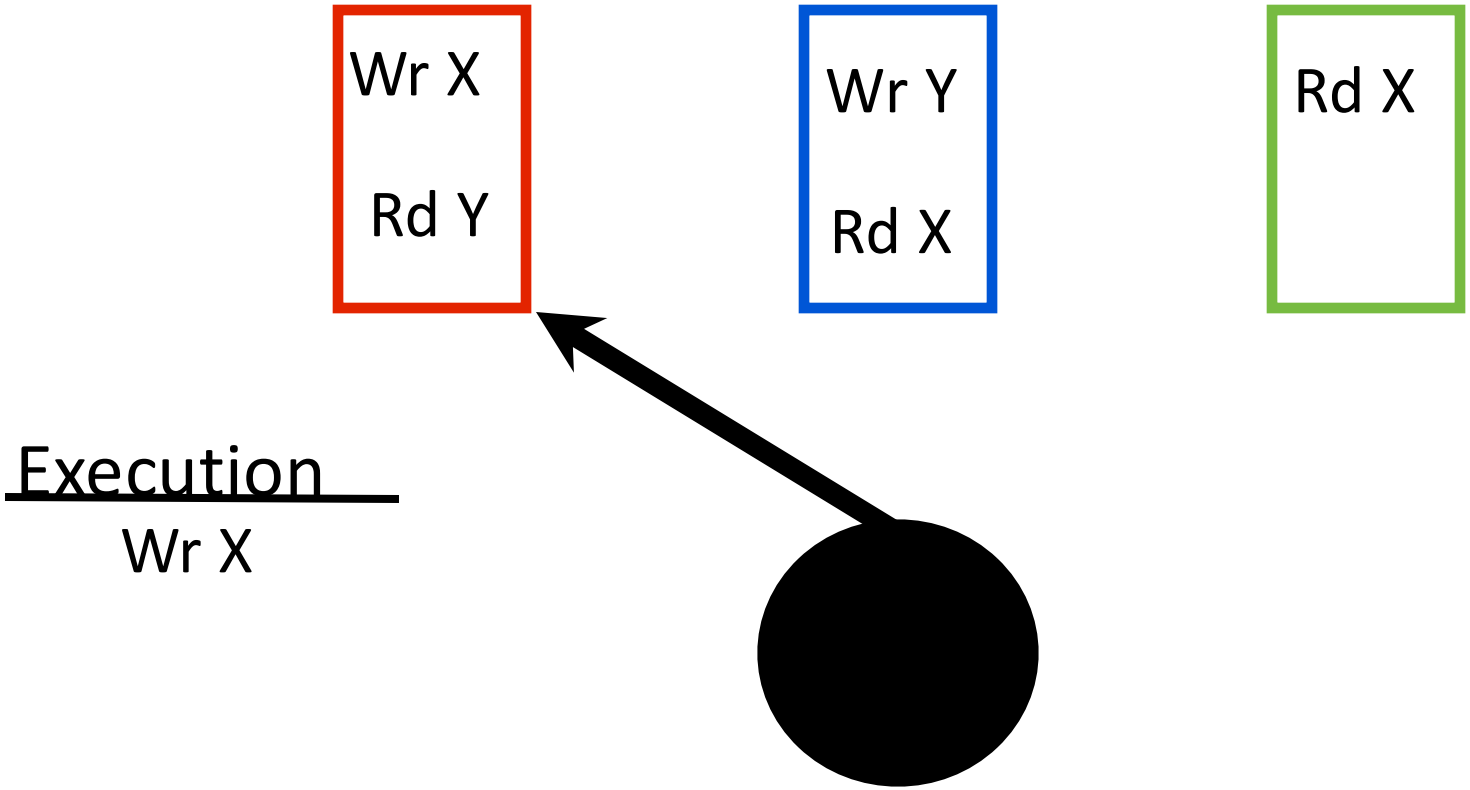
Invariant #1:
Instructions are executed in program order

Invariant #2:
All processors agree on a total order of executed instructions

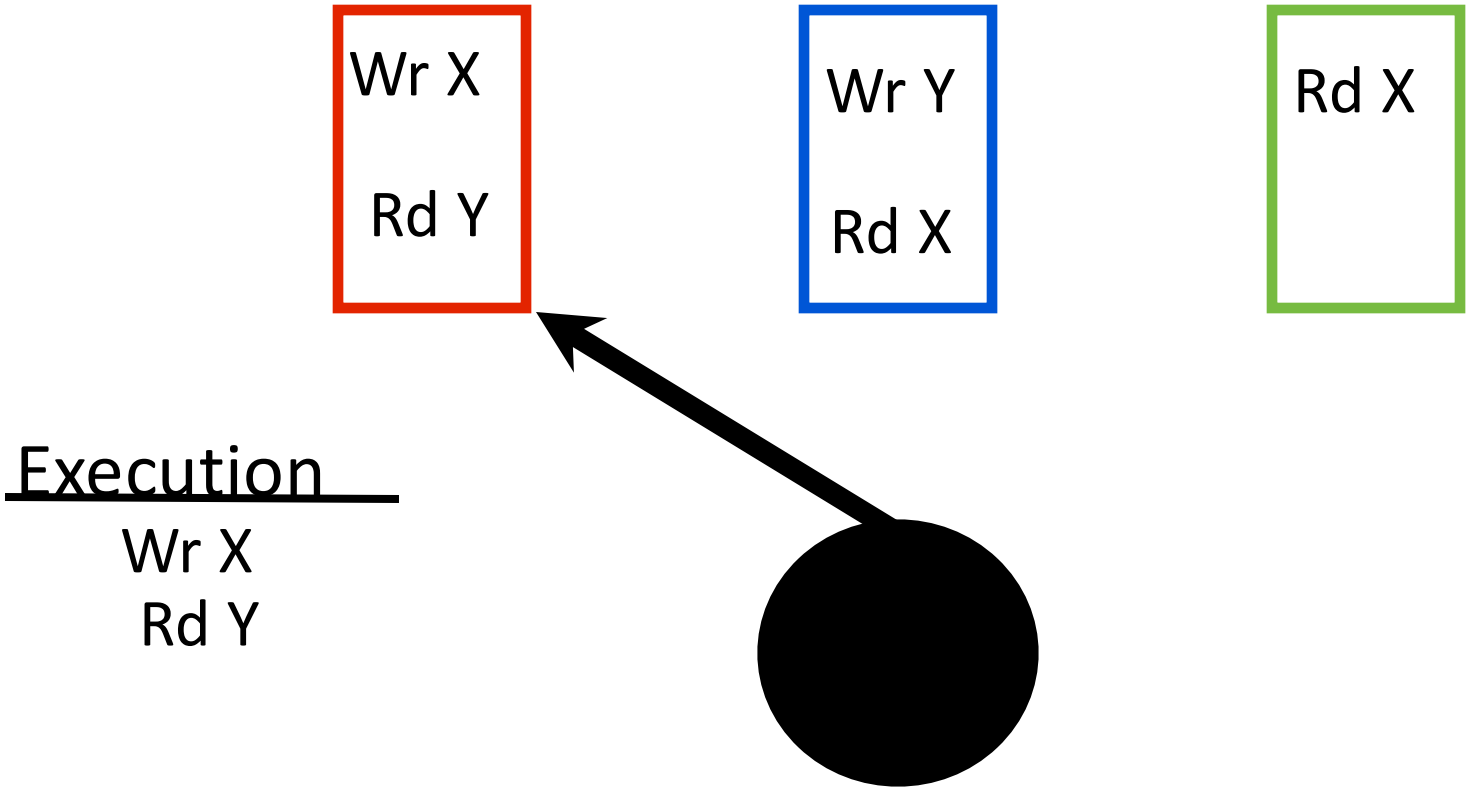
The SC “Switch”



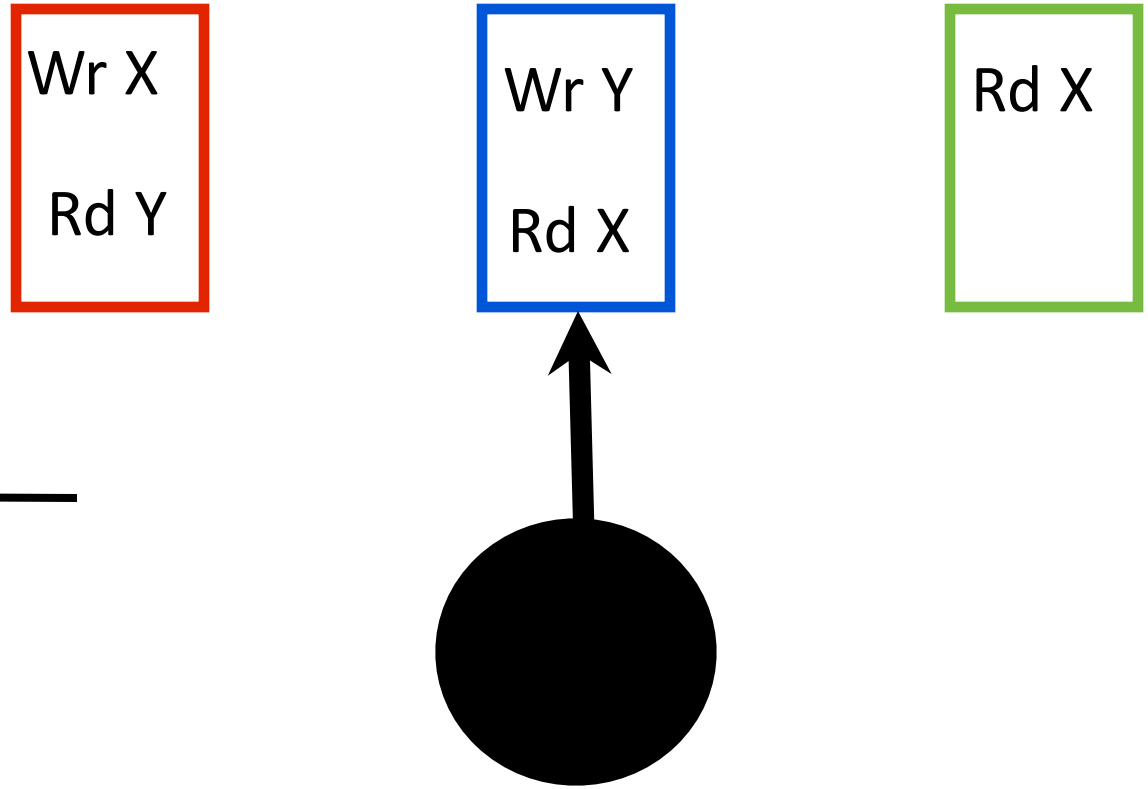
The SC “Switch”



The SC “Switch”



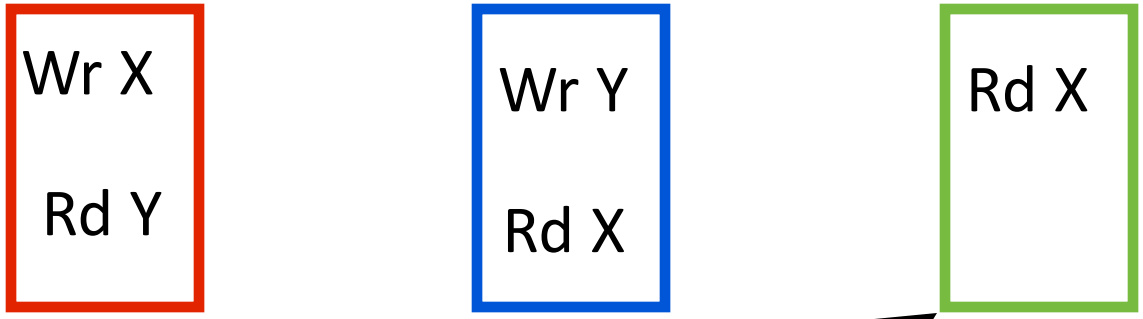
The SC “Switch”



Execution

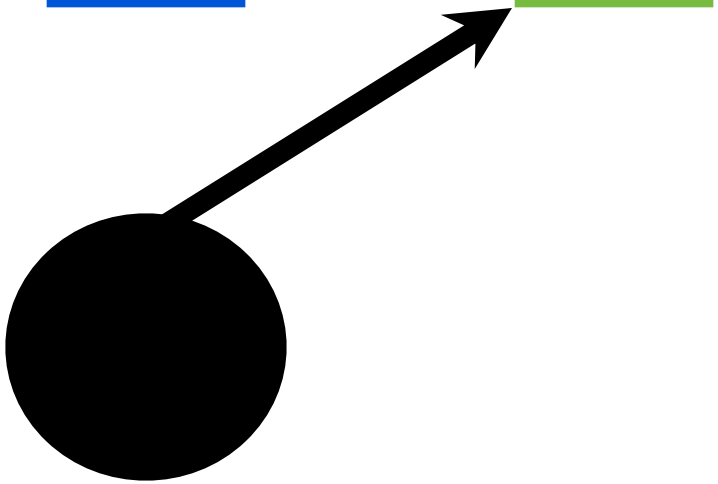
Wr X
Rd Y
Wr Y

The SC “Switch”

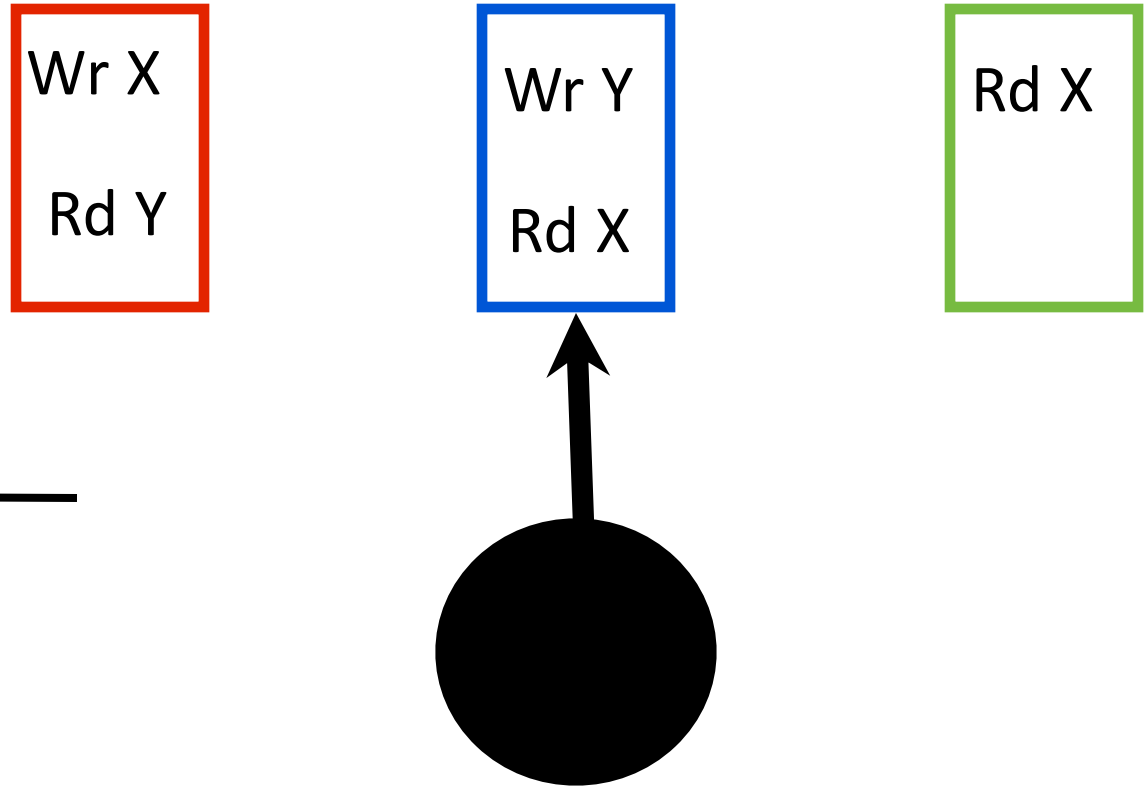


Execution

Wr X
Rd Y
Wr Y
Rd X



The SC “Switch”

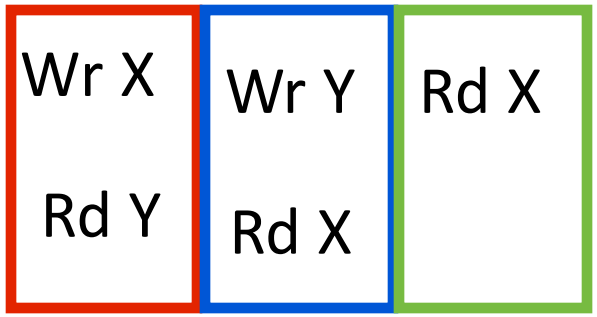


Execution

Wr X
Rd Y
Wr Y
Rd X
Rd X

Why is SC Important?

SC is the most complex model that we can ask **programmers** to think about.



Intuitive (SC)	Weird (not SC)
Wr X	Rd Y
Rd Y	Wr X
Wr Y	Rd X
Rd X	Rd X
Rd X	Wr Y

SC prohibits **all** reordering of instructions (Invariant 1)

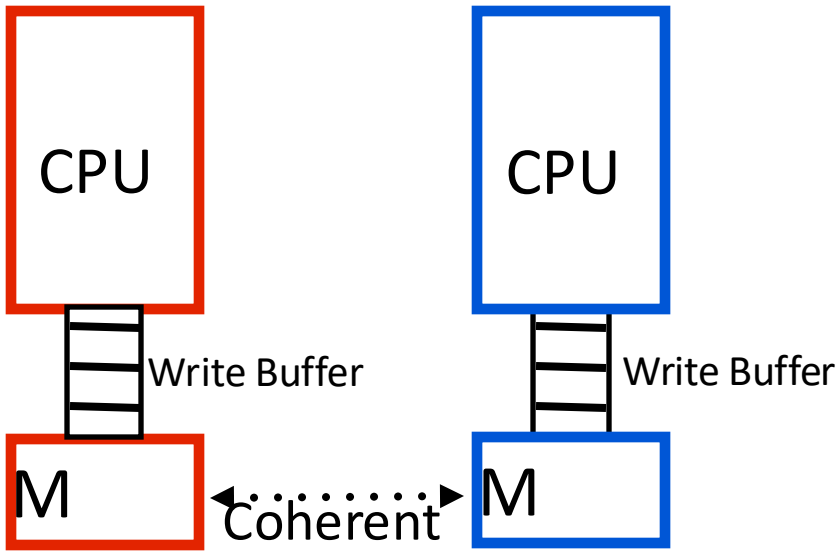
Real hardware does not enforce SC

The ARMv8 Memory Model:

The ARMv8 architecture employs a *weakly-ordered* model of memory. In general terms, this means that the order of memory accesses is not required to be the same as the program order for load and store operations. The processor is able to re-order memory read operations with respect to each other. Writes may also be re-ordered (for example, write combining) .As a result, hardware optimizations, such as the use of cache and write buffer, function in a way that improves the performance of the processor, which means that the required bandwidth between the processor and external memory can be reduced and the long latencies associated with such external memory accesses are hidden.

https://developer.arm.com/documentation/den0024/a/Memory-Ordering

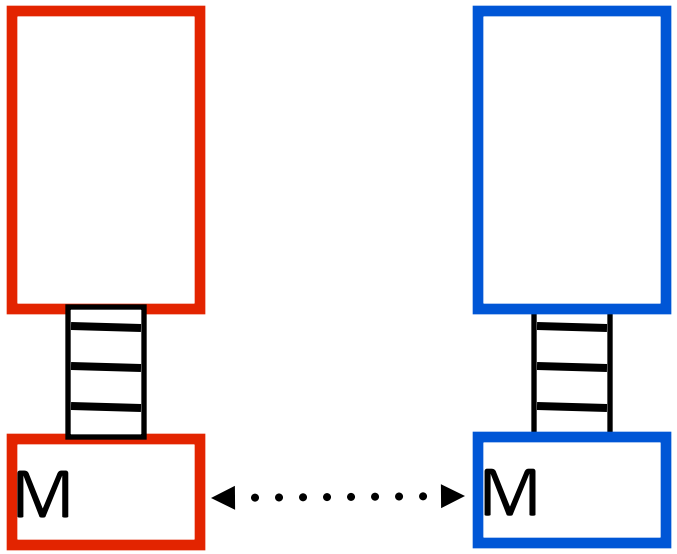
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



Program

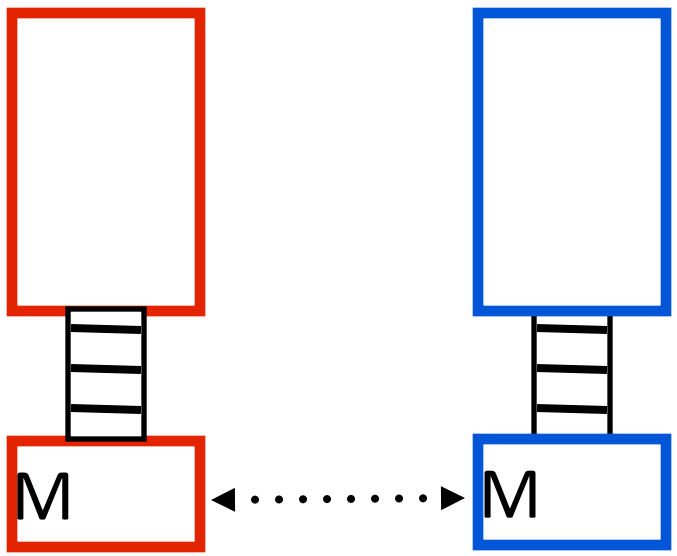
Initially X == Y == 0

X=1 Y=1

r1=Y r2=X

Is r1==r2==0
a valid result?

Reordering #1: Write Buffers



Program

Initially X == Y == 0

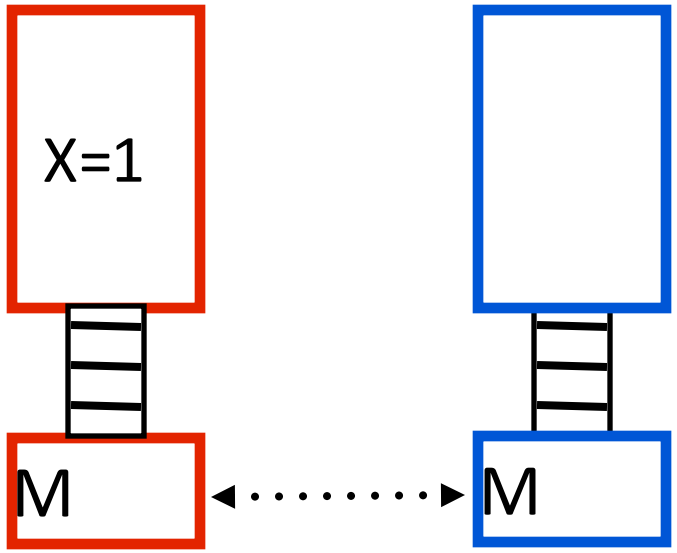
X=1 Y=1

r1=Y r2=X

Is r1==r2==0
a valid result?

r1 == r2 == 0 is **not** SC, but it can happen with write buffers

Reordering #1: Write Buffers



Program

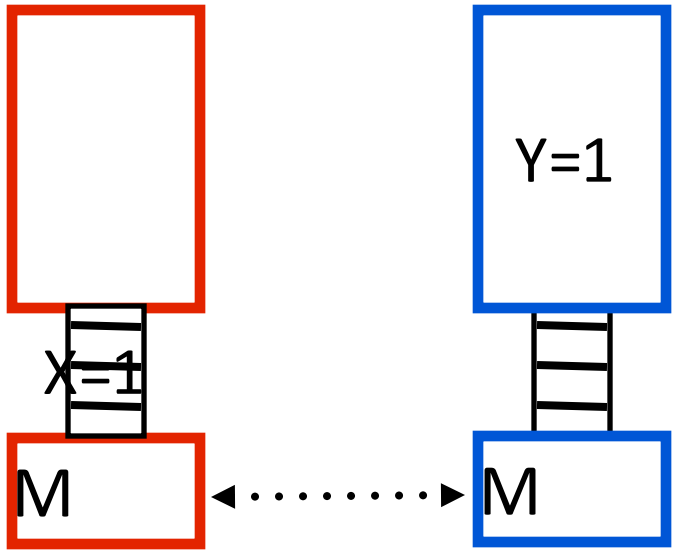
Initially X == Y == 0

Y=1

r1=Y r2=X

Execution

Reordering #1: Write Buffers



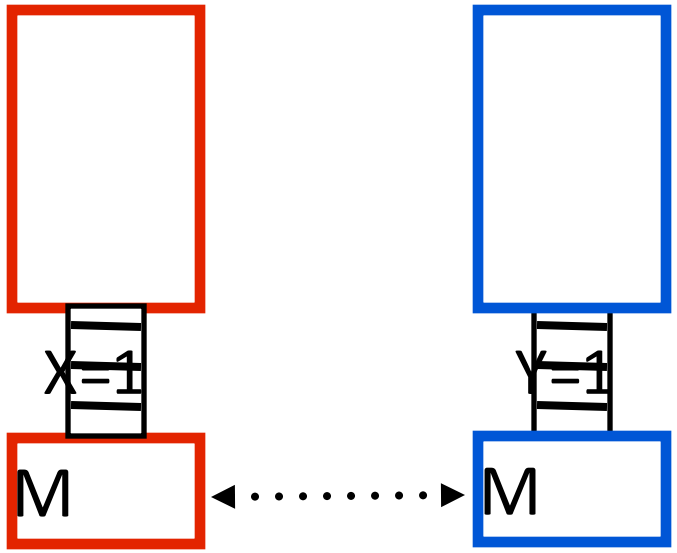
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



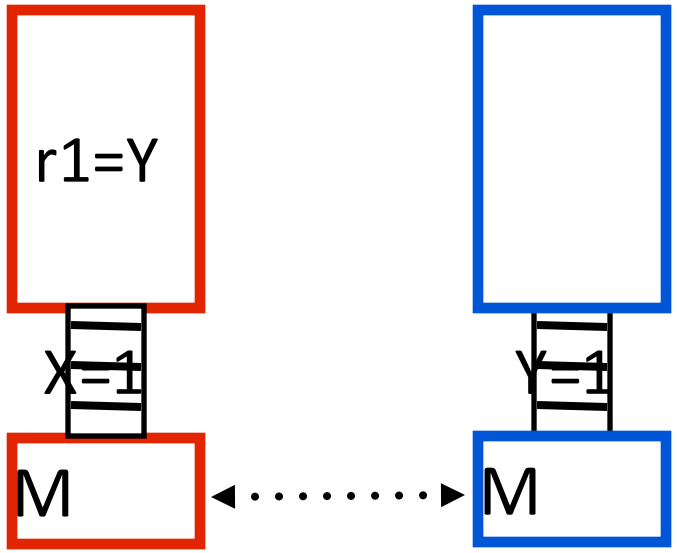
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



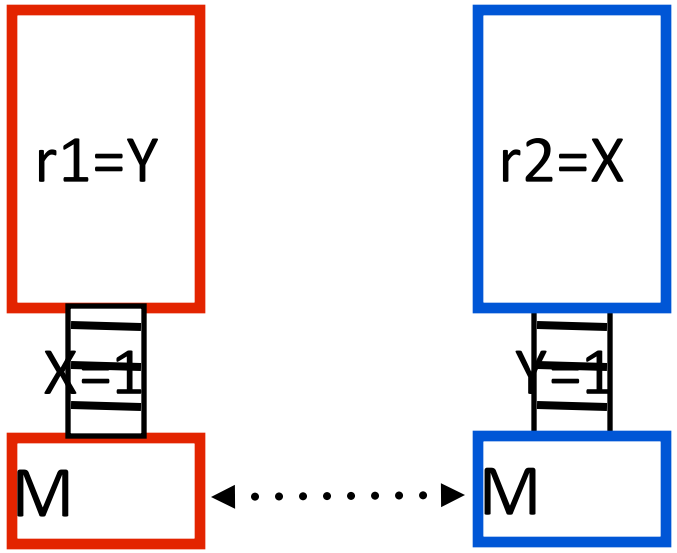
Program

Initially X == Y == 0

r2=X

Execution

Reordering #1: Write Buffers

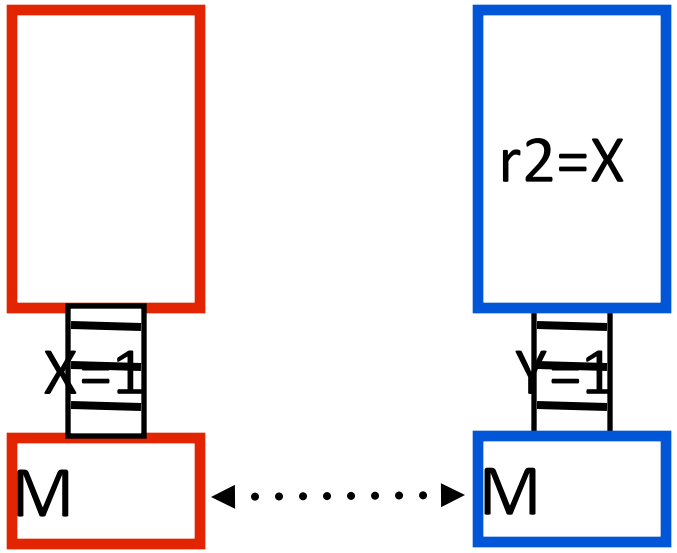


Program

Initially X == Y == 0

Execution

Reordering #1: Write Buffers



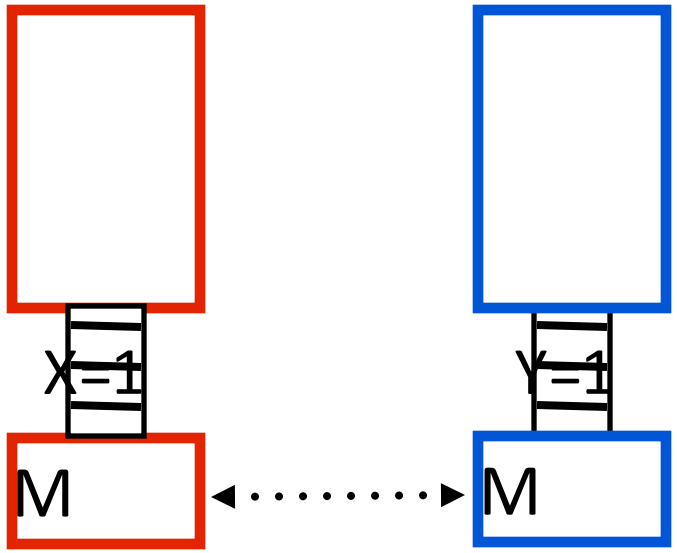
Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

Reordering #1: Write Buffers



Program

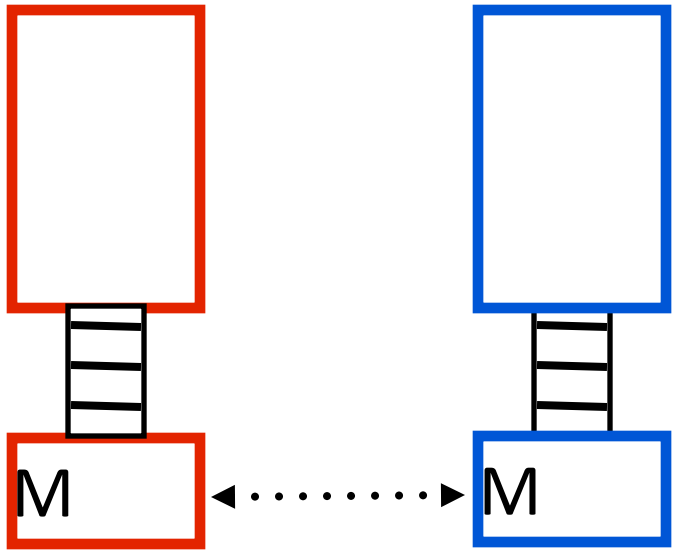
Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

Reordering #1: Write Buffers



WBs let reads finish before older writes

Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

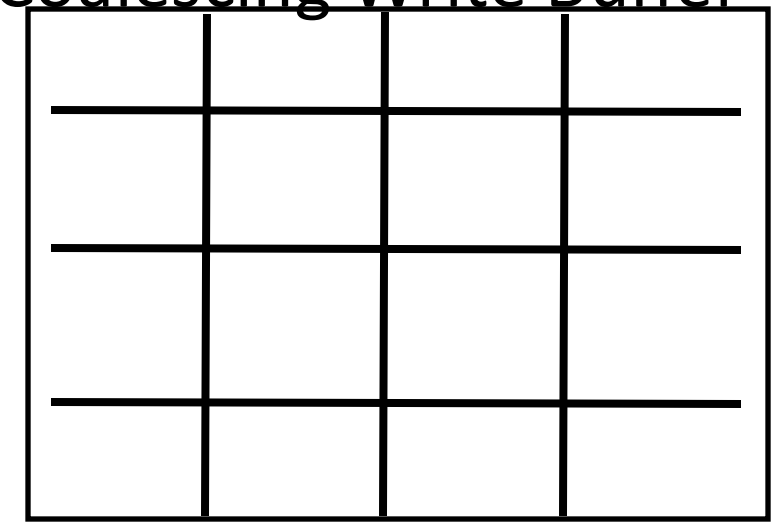
r2=X [r2 <- 0]

X=1

Y=1 (Not SC!)

Reordering #2: Write Combining

Coalescing Write Buffer



4 word cache line

Program

X,Z in same $ line

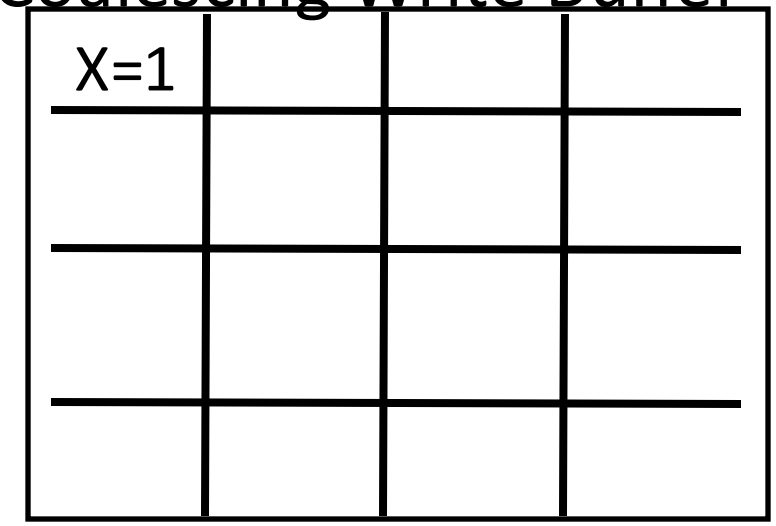
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

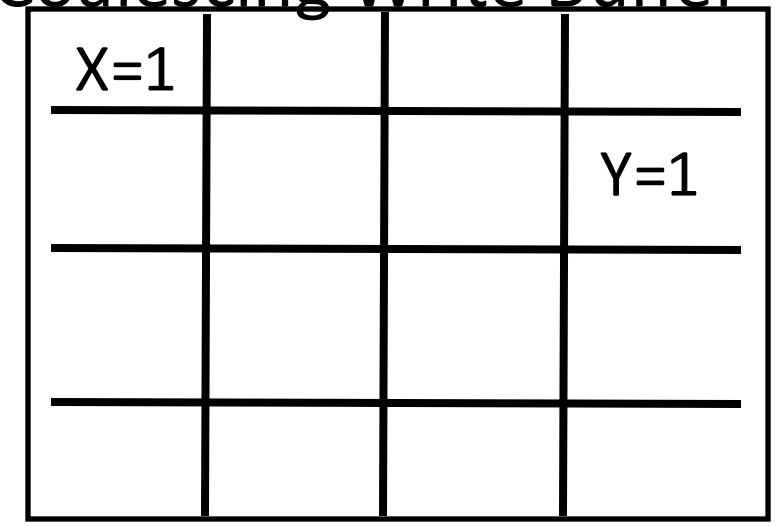
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

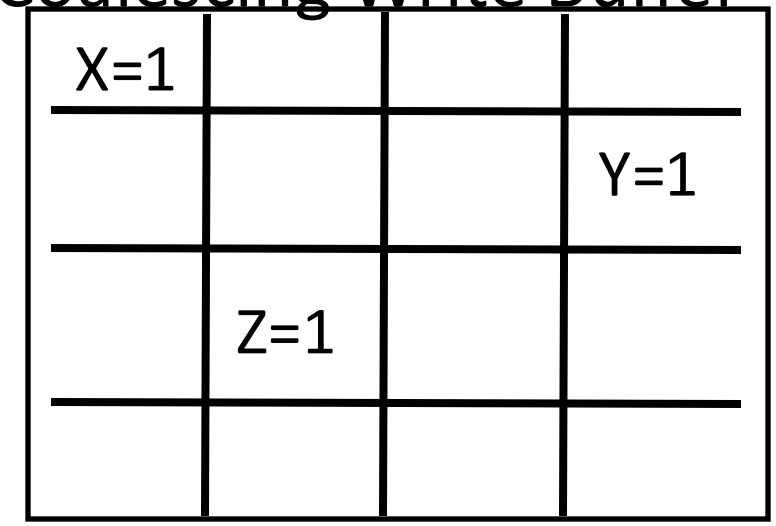
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

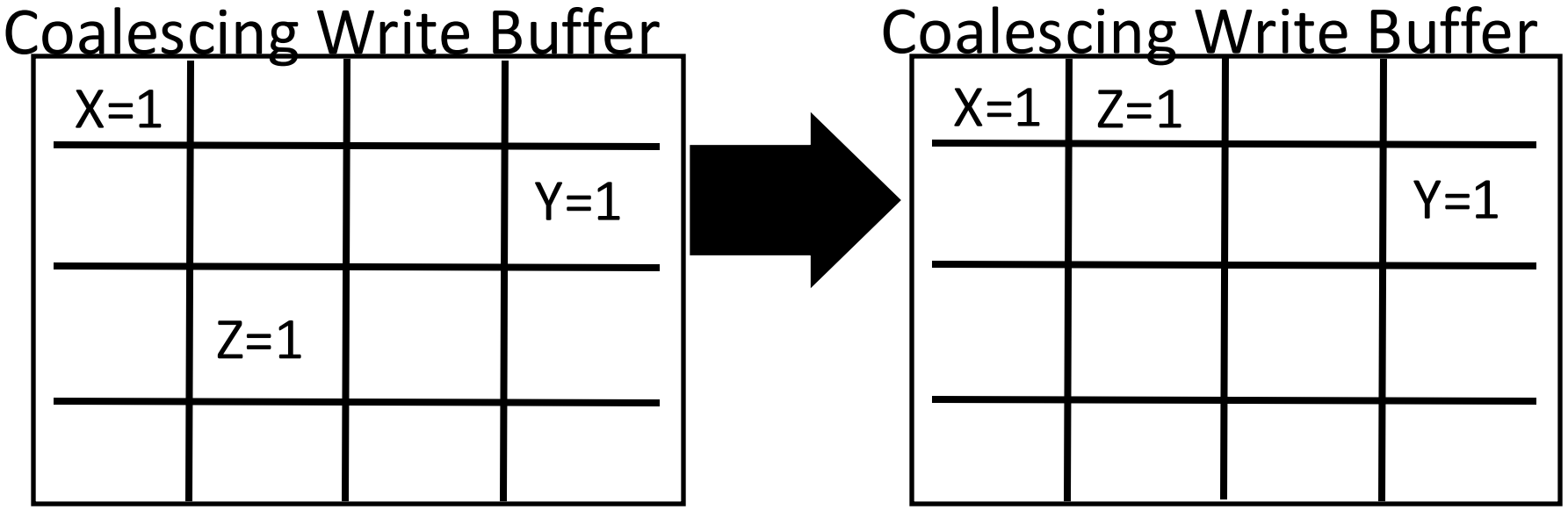
X,Z in same $ line

X=1

Y=1

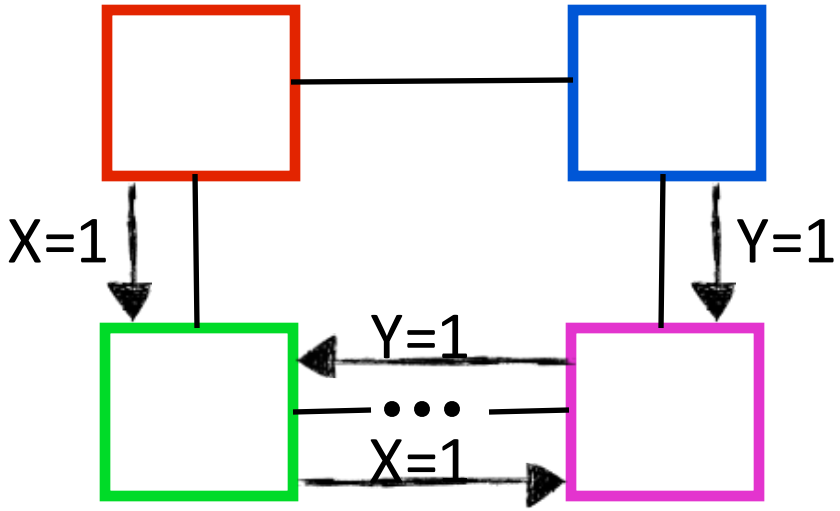
Z=1

Reordering #2: Write Combining



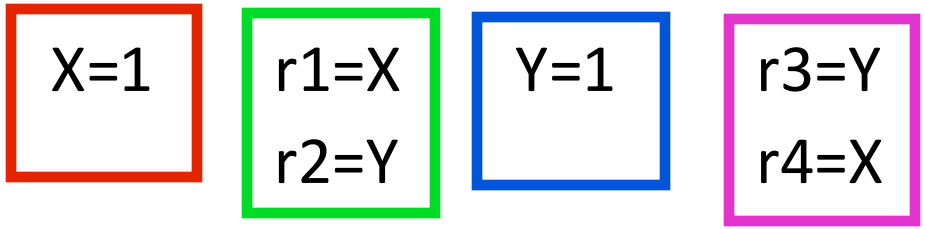
Combining the write to X & Z saves bandwidth,
but **reorders** Z=1 and Y=1

Reordering #3: Interconnect



Variable time cost traversing routed on-chip network

4-threaded program



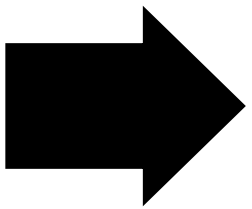
Execution

X=1
Y=1
r1=X [r1 <- 1]
r2=Y [r2 <- 0]
r3=Y [r3 <- 1]
r4=X [r4 <- 0]

Reordering #4: Compilers

```
X = 0
for (1 .. 100)
    X = 1
    print X
```

X = 0




```
X = 1
for (1 .. 100)
    print X
```

X = 0

The compiler hoists the write out of the loop, permitting new (non-SC) results (e.g., “1 0 0 0 0 0 0 0...”)

When is an Execution Not SC?

When a memory operation happens before itself

Execution

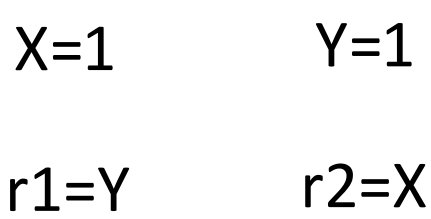
r1=Y [r1 <- 0]

r2=X [r2 <- 0]

X=1

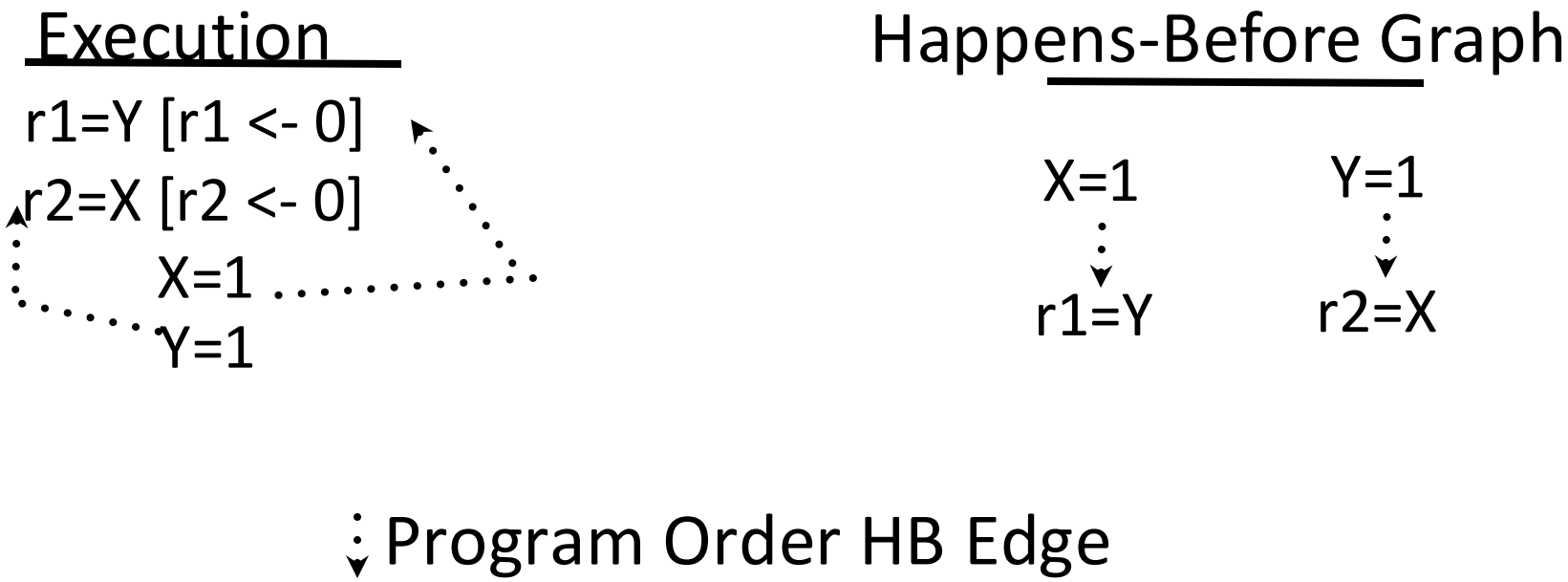
Y=1

Happens-Before Graph



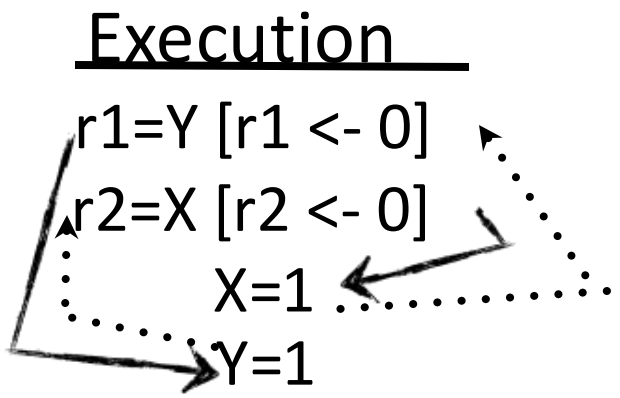
When is an Execution Not SC?

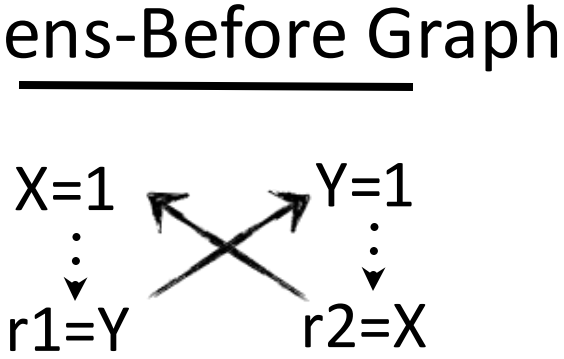
When a memory operation happens before itself



When is an Execution Not SC?

When a memory operation happens before itself



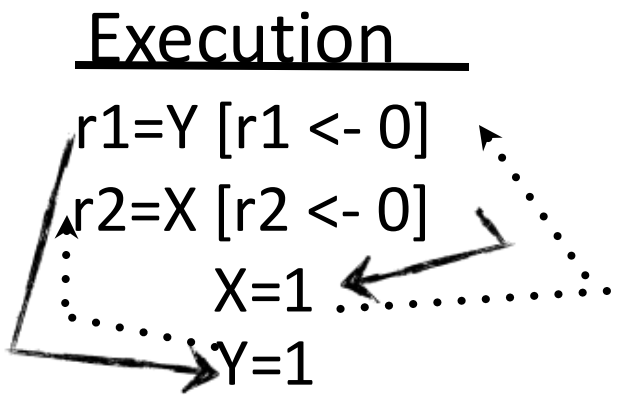


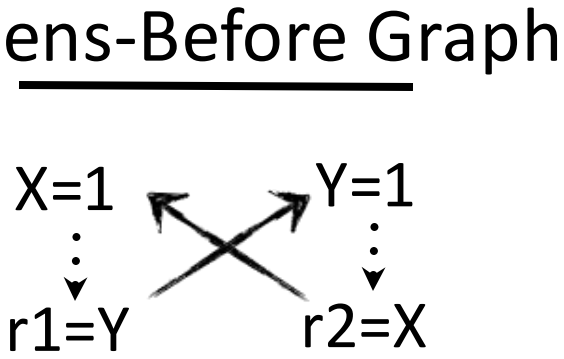
Program Order HB Edge

Causal Order HB Edge

When is an Execution Not SC?

When a memory operation happens before itself

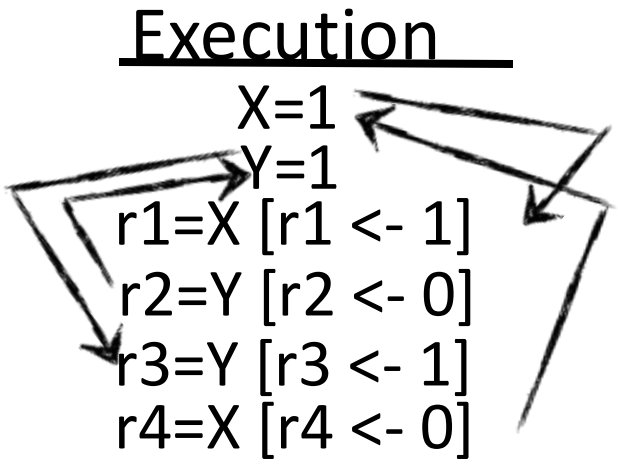


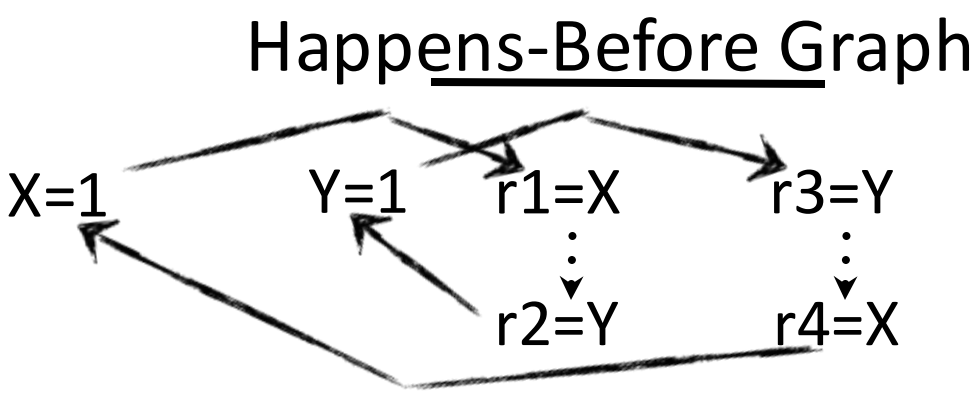


If there is a cycle in the happens-before graph, the execution is not SC

When is an Execution Not SC?

When a memory operation happens before itself





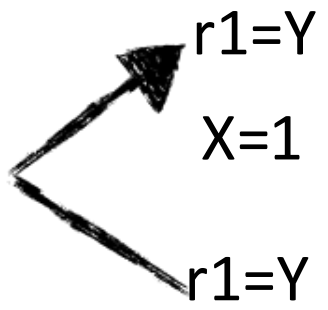
If there is a cycle in the happens-before graph, the execution is not SC

Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC

x86-TSO (intel x86s)

"The Write Buffer Memory Model"



Relaxes W->R order

Total **S**tore **O**rder - loads may complete before older stores to different locations complete.

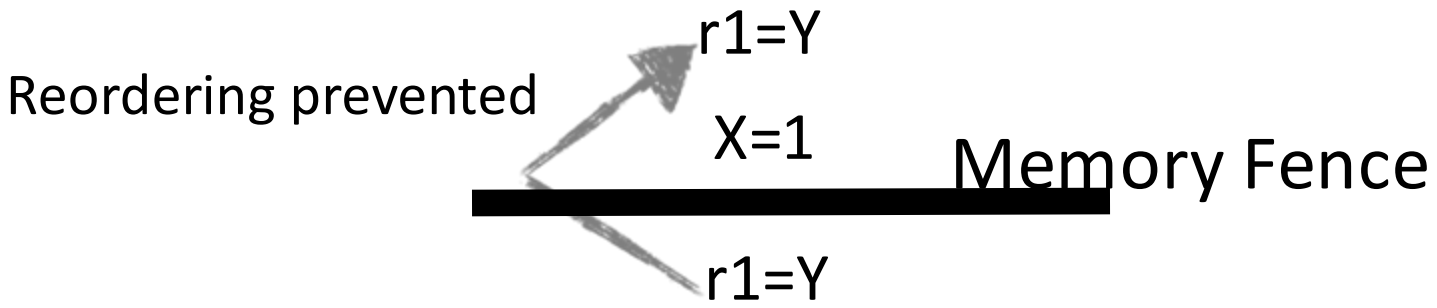
Implementing Synchronization for Weak Memory Models

- What does synchronization have to do to prevent SC violations?
  - Flush WB, prevent coalescing/bypassing, impose ordering in network, prevent compiler reorderings
- What does synchronization have to do to prevent other kinds of problems?
  - Enforce mutually exclusive execution by different threads of critical region, force threads to wait at barriers, enforce wait/notify discipline

Synchronization and its Implementation

Synchronization Can Prevent Operation Reordering

Memory fences are one type of synchronization

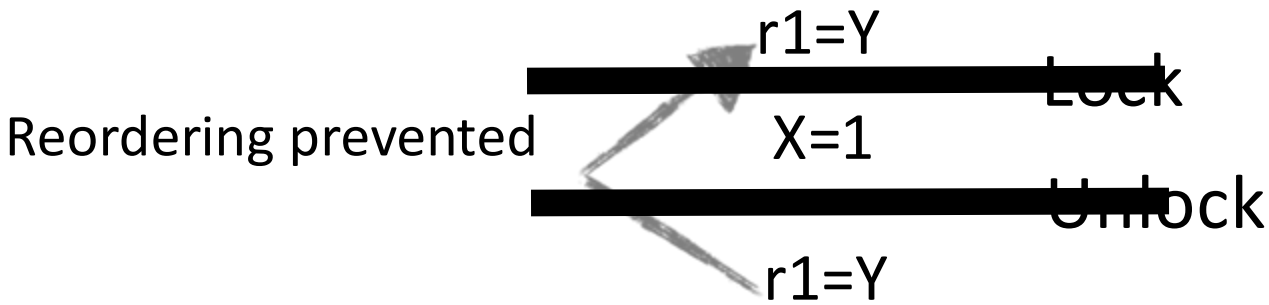


Fence implementation depends on reordering implementation

We will see later in this lecture why reordering matters so much.

Synchronization For Real Programmers

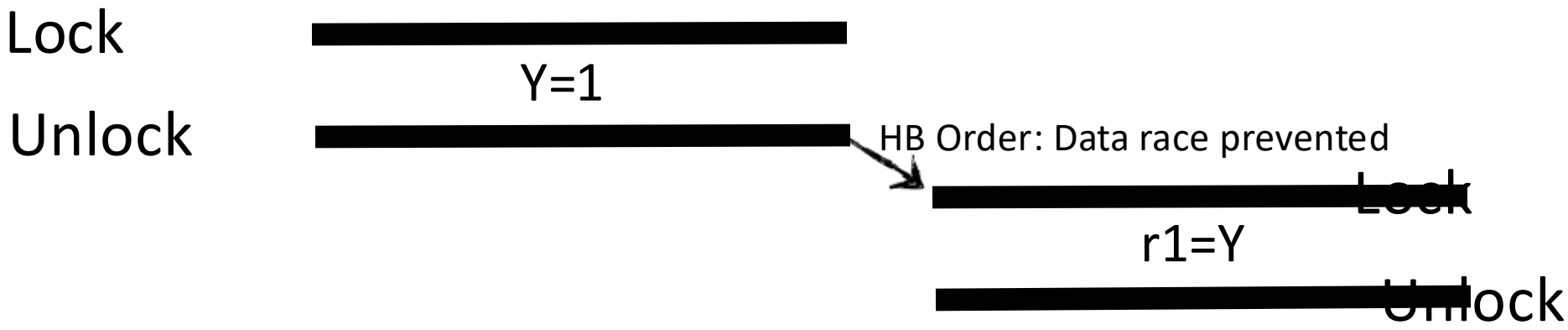
Memory fences are wrapped up in locks, etc.



Direct use of fences can be tricky and you will usually use a library

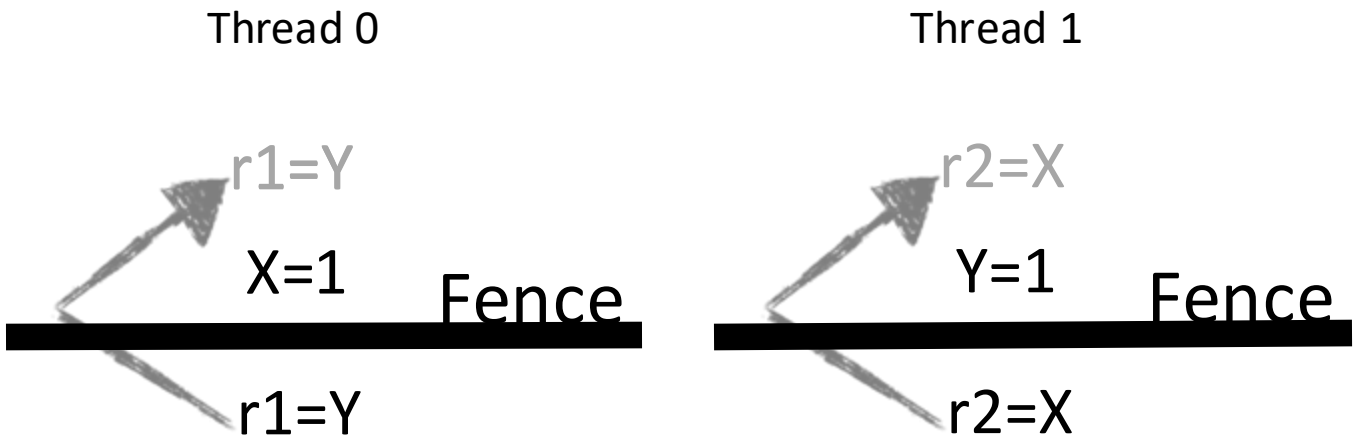
Data Races

Synchronization imposes happens-before on otherwise unordered operations



Data Race: Unordered operations to the same memory location, at least one write.

Fences are for (Preventing Re-)Ordering to Avoid Data Races & Ensure Correct Executions



We will see later that this program can produce very strange results if not sychronized

Fences are for (Preventing Re-)Ordering to Avoid Data Races & Ensure Correct Executions

Thread 0

r1=X
r1++
X=r1

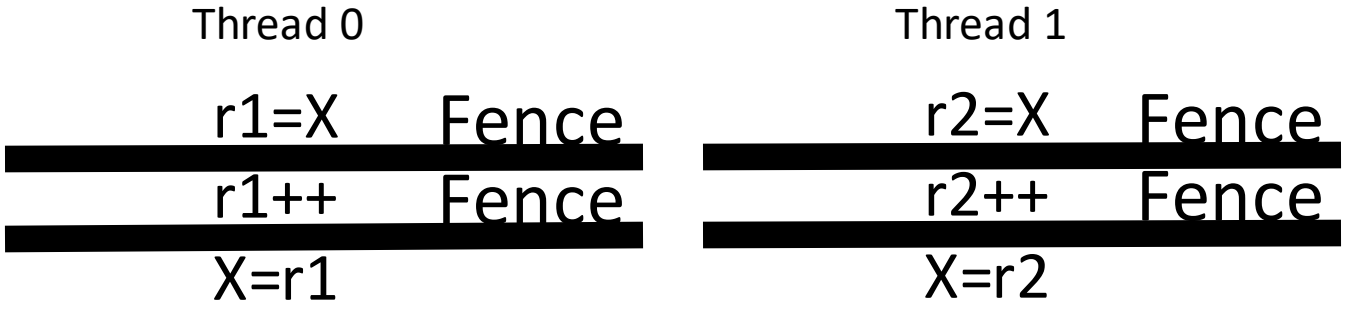
Thread 1

r2=X
r2++
X=r2

What happens with this program? Where can we put the fence?

Fence

Fences are for (Preventing Re-)Ordering to Avoid Data Races & Ensure Correct Executions



How about fences everywhere? Does this fix our problem?

Some programs also require *atomicity*

Thread 0

```
r1=X
r1++
X=r1
```

Thread 1

```
r2=X
r2++
X=r2
```

Defining Atomicity:
All-or-nothing behavior of critical regions.

Execution = serialization of crit. regs.

Fences don't provide atomicity

Some programs also require *atomicity*

Serialization #1

```
r1=X
r1++
X=r1
```

```
r2=X
r2++
X=r2
```

Serialization #2

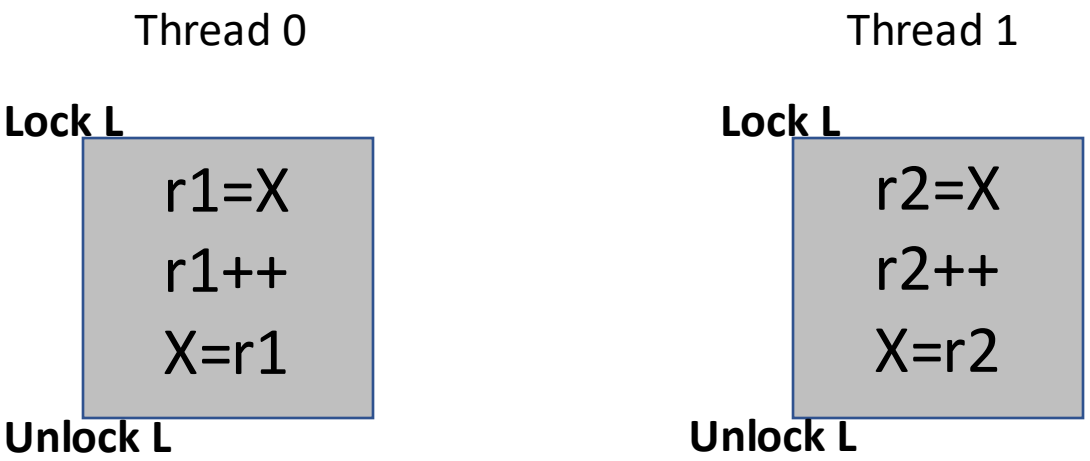
```
r2=X
r2++
X=r2
```

```
r1=X
r1++
X=r1
```

Defining Atomicity:
All-or-nothing behavior of critical regions.

Execution = serialization of crit. regs.

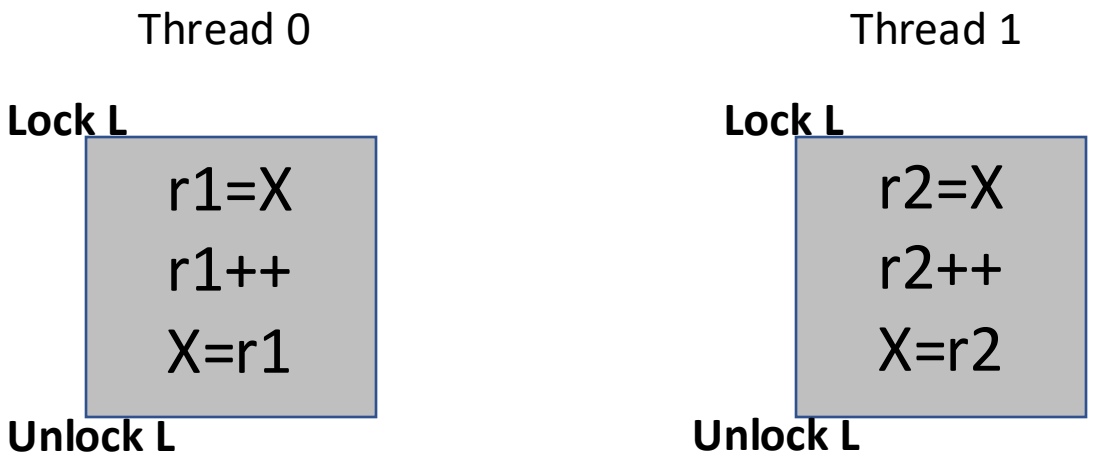
Mutual exclusion (mutex) locks enforce *atomicity* (and ordering)



Lock Behavior:
A thread *acquires* a lock L, does stuff while *holding* L, and then *releases* lock L.

If a thread tries to acquire L while L is held, the thread keeps trying to acquire L until L is *unheld*, when its attempt to acquire succeeds.

SpinLock is one lock implementation




```
spinlock(L){
  while(__sync_bool_compare_and_swap(&L,0,1) == 0 ){
    /*do nothing; pause here on some systems*/
  }
}
unlock(L){ L = 0; __sync_synchronize(); /*mem fence*/ }
```

SpinLock is one lock implementation

```
spinlock(L){
  while(  sync bool compare and swap(&L,0,1) == 0 ){
    /*do nothing; pause here on some systems*/
  }
}
unlock(L){ L = 0; __sync_synchronize(); /*mem fence*/ }
```

Turtles all the way down?

SpinLock is one lock implementation

```
spinlock(L){
  while(__sync_bool_compare_and_swap(&L,0,1) == 0 ){
    /*do nothing; pause here on some systems*/
  }
}
unlock(L){ L = 0; __sync_synchronize(); /*mem fence*/ }
```

```
175b: 48 8b 02        mov    (%rdx),%rax //load L into %rax
175e: 48 8d 48 01     lea    0x1(%rax),%rcx //add 1 to %rax, into %rcx
1762: f0 48 0f b1 0a  lock cmpxchg %rcx,(%rdx) //compare & exchange
1767: 75 f2           jne    175b //loop to mov if cmpxchg fails
```

SpinLock is one lock implementation

```
spinlock(L){
  while(  sync bool compare and swap(&L,0,1) == 0 ){
    /*do nothing; pause here on some systems*/
  }
}
unlock(L){ L = 0; __sync_synchronize(); /*mem fence*/ }
```

```
1762: f0 48 0f b1 0a  lock cmpxchg %rcx,(%rdx)
//if (%rdx) == %rax{   (%rdx) = %rcx  }
```

Implemented directly in the machine microarchitecture. Even if multiple threads executing, hardware guarantees *no inter-thread interactions*

SpinLock is one lock implementation

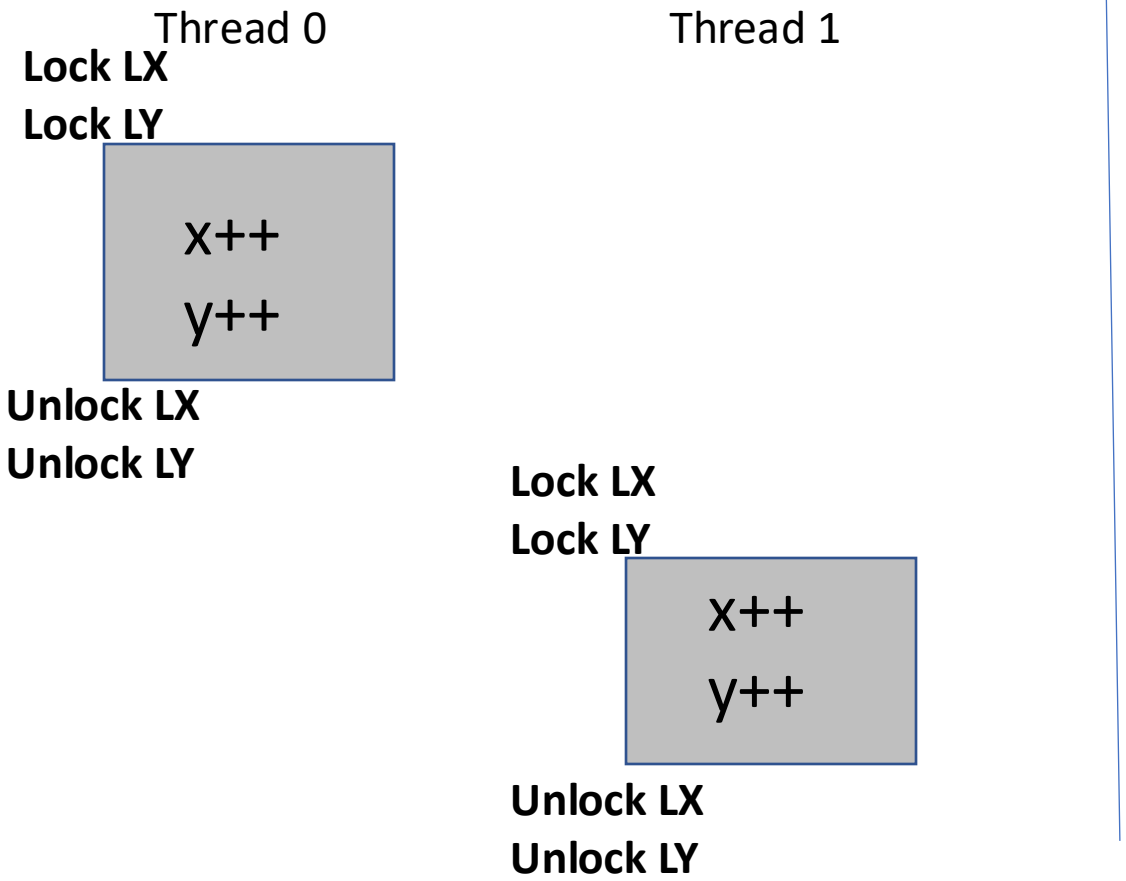
```
spinlock(L){
  while(__sync_bool_compare_and_swap(&L,0,1) == 0 ){
    /*do nothing; pause here on some systems*/
  }
}
unlock(L){ L = 0; __sync_synchronize(); /*mem fence*/ }
```

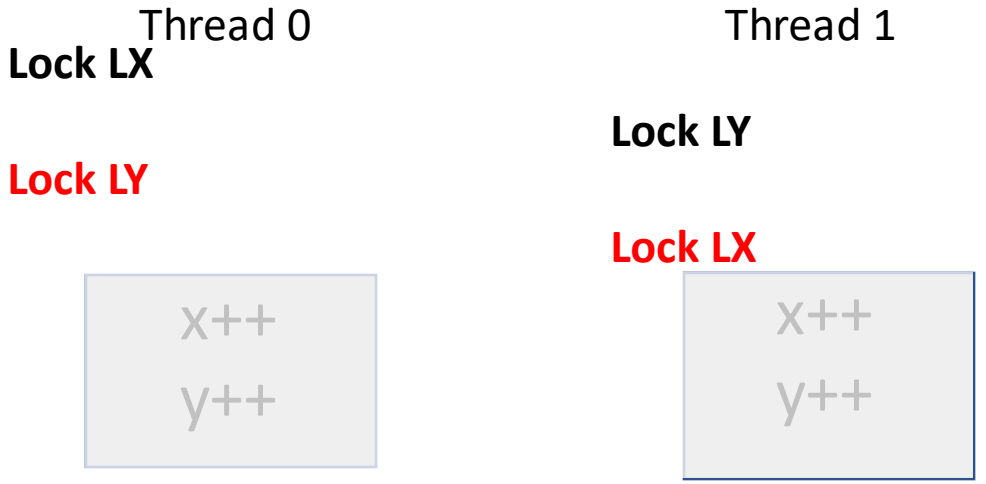
```
1890:  0f ae f0    mfence
```

Fence

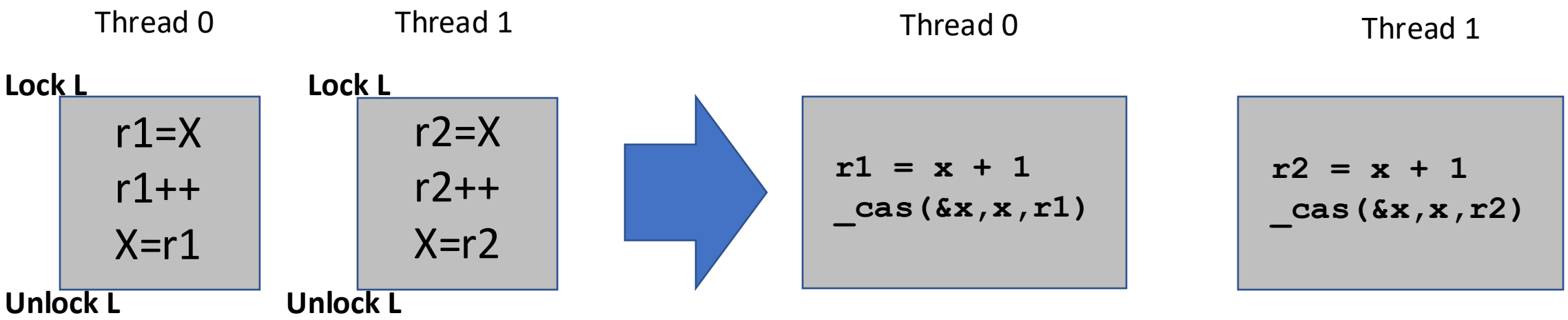
Lock ordering matters

Lock Ordering:
If you manipulate more than one piece of data in a critical region, you will need to acquire the locks in the same order for all critical regions or face **deadlock**





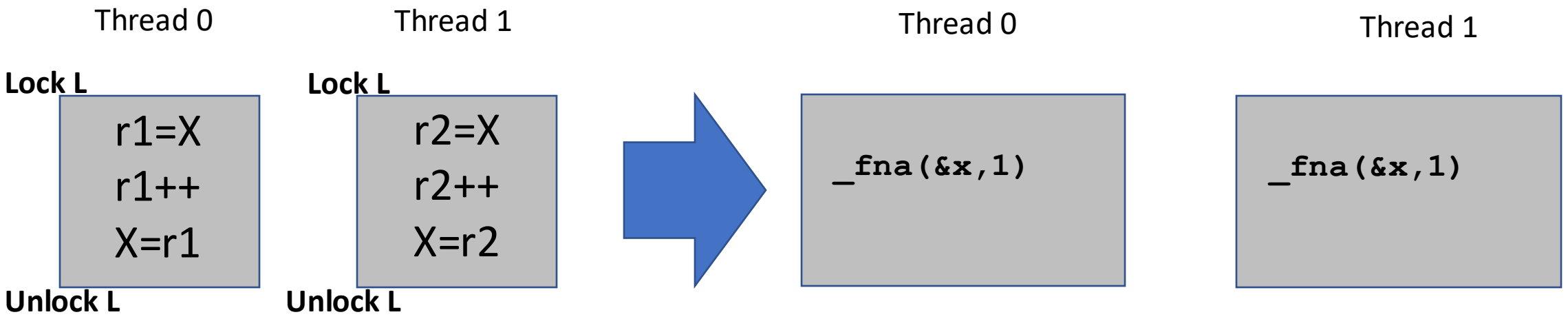
Directly Using Compare and Swap



How general is a CAS operation for implementing critical regions that need to execute atomically? What are the limitations on a CAS operation?

Fetch and Add – Further Specializing Atomics

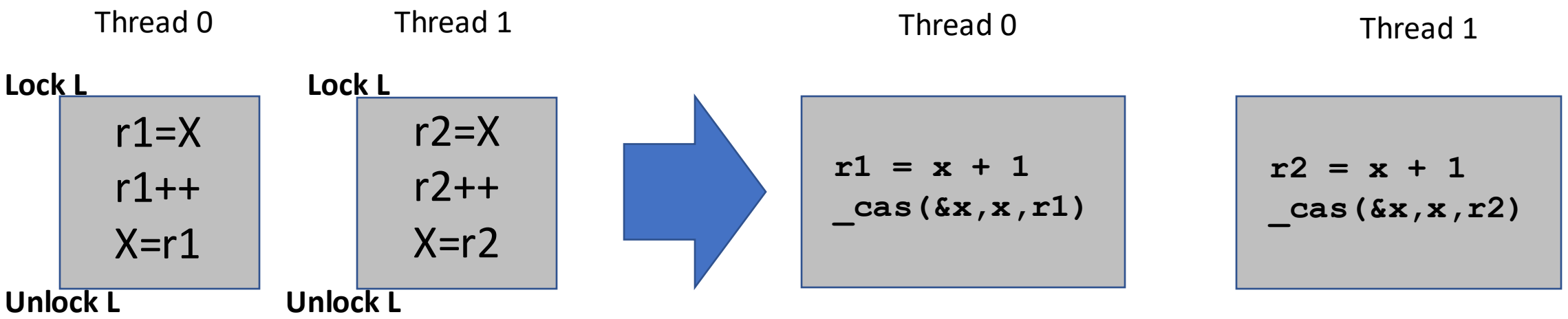
```
__sync_fetch_and_add(x,1);
```




```
1707: f0 48 83 04 d0 01    lock addq $0x1,(%rax,%rdx,8)
```

How much *less* general than compare and swap?

Transactional Memory – Further *Generalizing* Atomics



Limited by *single location* that can be updated using a CAS. What if we want to update 3 (or *n*) different locations (without using a lock)?

Transactional Memory – Further *Generalizing* Atomics

Thread 0

Lock L

```
r1=X
r1++
X=r1
r2=Y
Y++
Y=r2
```

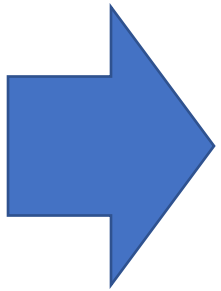
Unlock L

Thread 1

Lock L

```
r3=X
r3++
X=r3
r2=Y
Y++
Y=r2
```

Unlock L



Thread 0

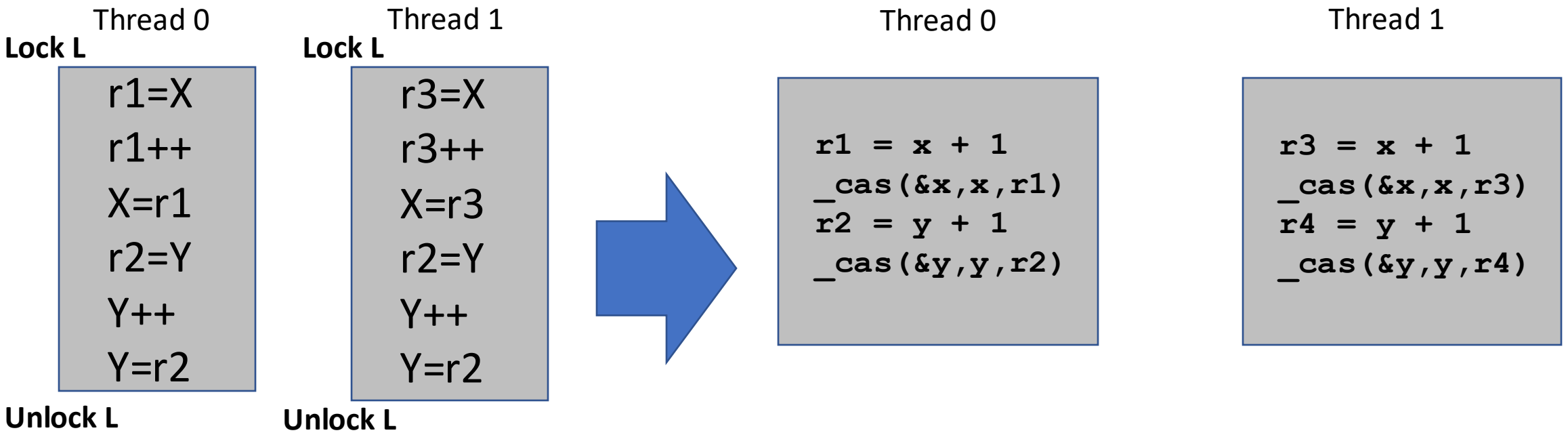
```
r1 = x + 1
_cas(&x,x,r1)
r2 = y + 1
_cas(&y,y,r2)
```

Thread 1

```
r3 = x + 1
_cas(&x,x,r3)
r4 = y + 1
_cas(&y,y,r4)
```

How about using multiple CAS operations?

Transactional Memory – Further *Generalizing* Atomics



How about using multiple CAS operations?

Problem: Need atomicity across CAS ops.

Transactional Memory: Atomicity for “n-CAS”

Thread 0

```
xbegin()
r1 = x + 1
r2 = y + 1
x = r1
y = r2
xend()
```

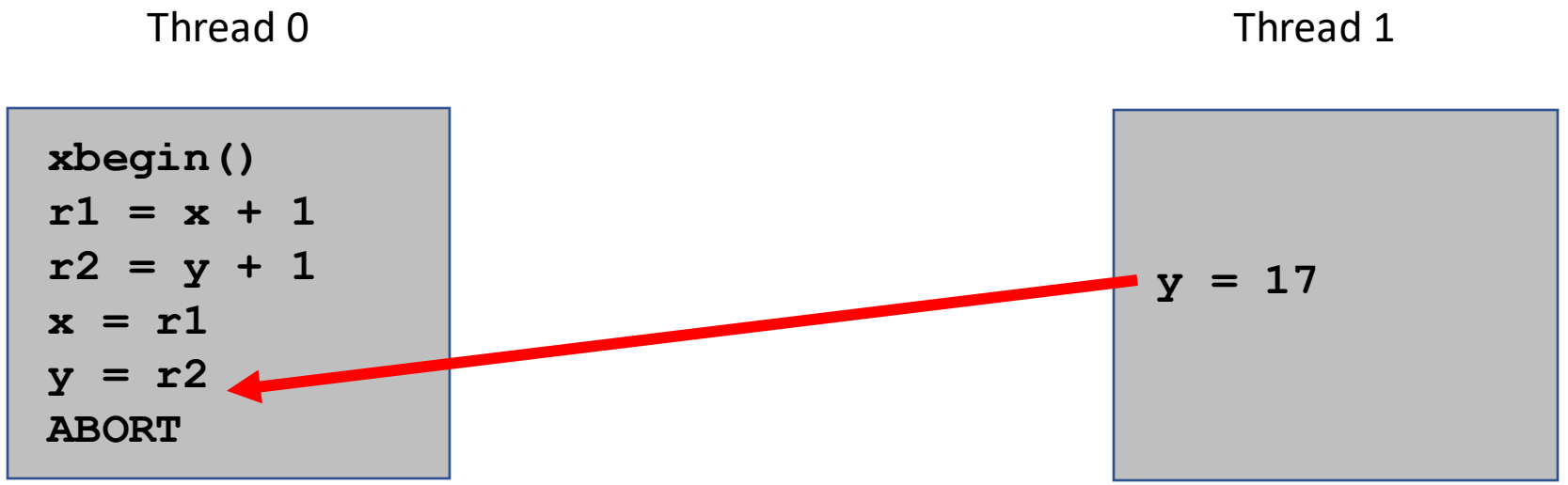
xbegin() starts a **transaction**
xend() ends the transaction started by the most recent xbegin()

Thread 1

```
xbegin()
r1 = x + 1
r2 = y + 1
x = r1
y = r2
xend()
```

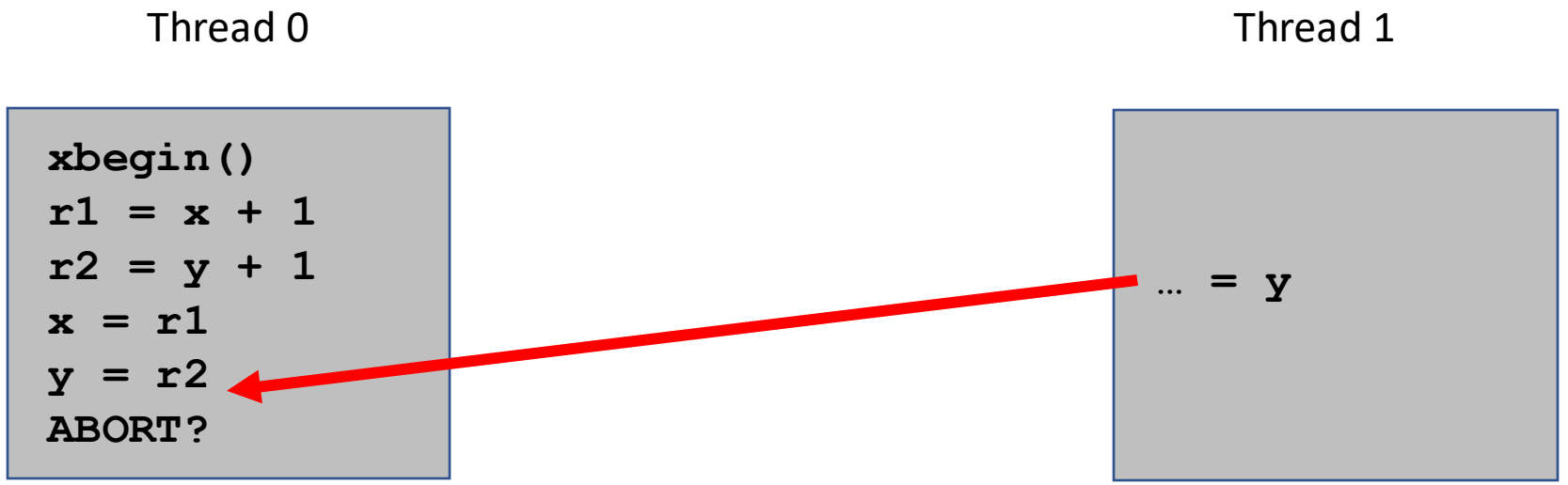
Transaction *attempts* to execute atomically, as if protected by a lock

Transactional Memory: Atomicity for “n-CAS”



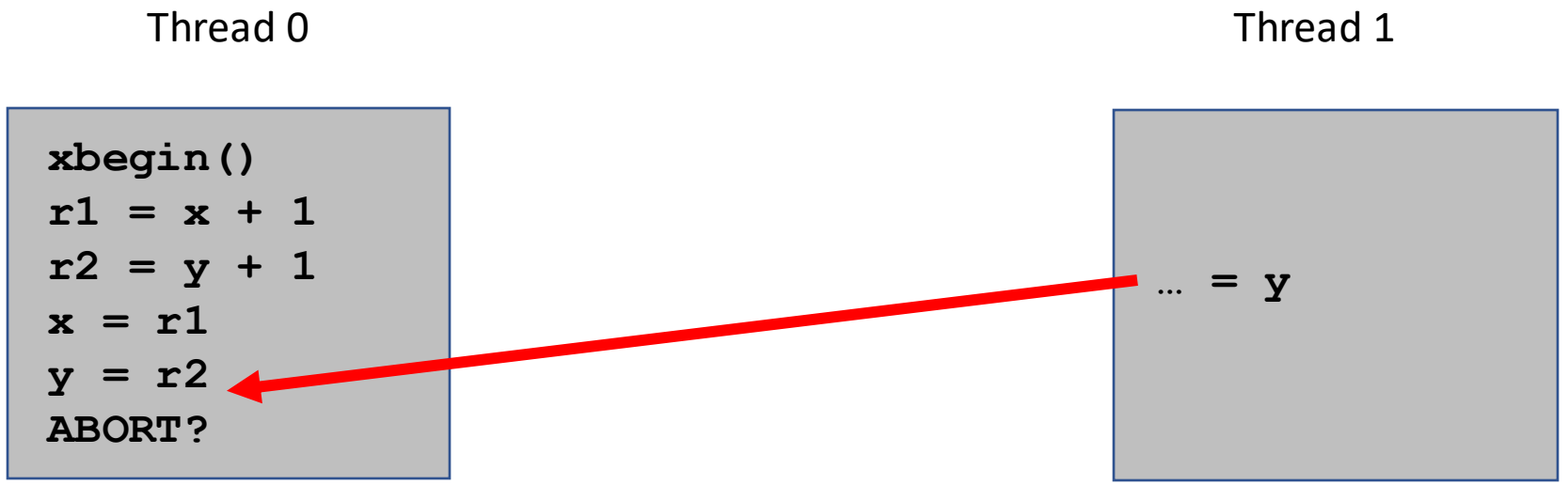
Transaction *aborts* if another thread accesses a location accessed in transaction (or if explicitly aborted)

Transactional Memory: Atomicity for “n-CAS”



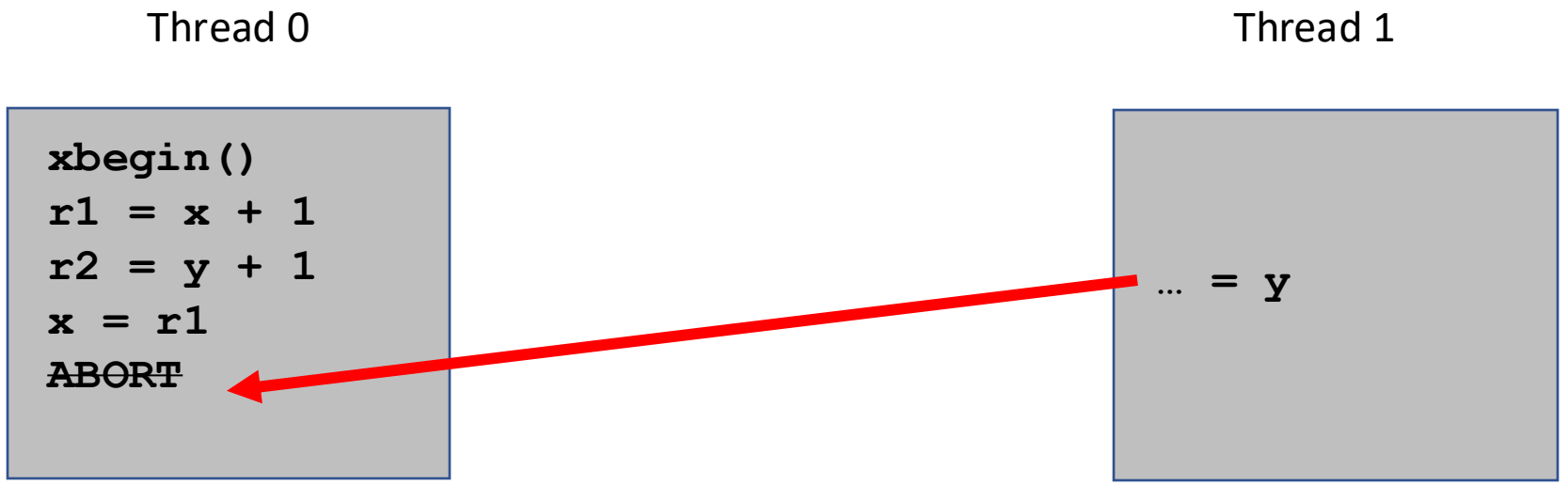
Transaction *aborts* if another thread
accesses a location accessed in transaction
(or if explicitly aborted)

Transactional Memory: Atomicity for “n-CAS”



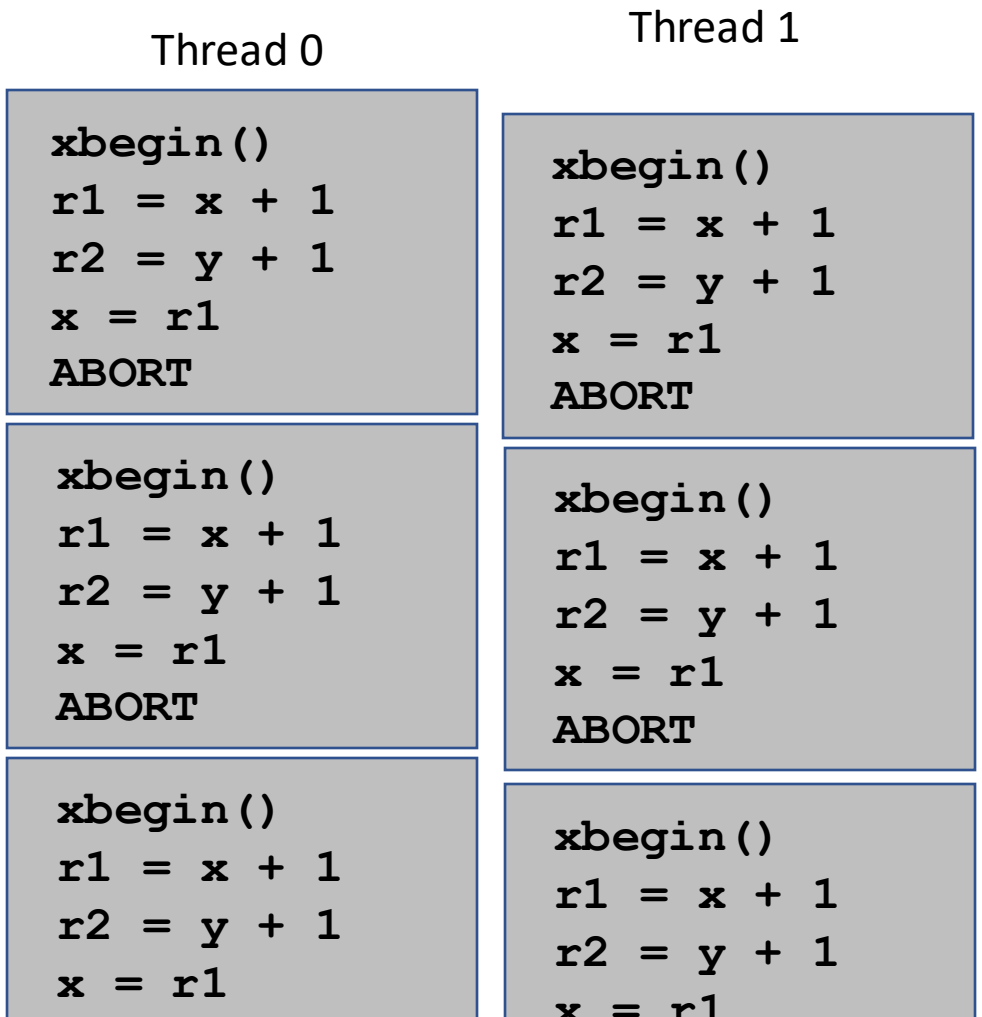
Transaction *aborts* if another thread **reads** a location written by the transaction or **writes** a location accessed by the transaction (“Conflicting” accesses)

Transactional Memory: Atomicity for “n-CAS”



Reads don’t conflict and
transactions can read-share data

What do we do if we have repeated aborts?



These threads are *contending for memory locations* causing repeated aborts.

How to deal with contention in a transactional memory system?

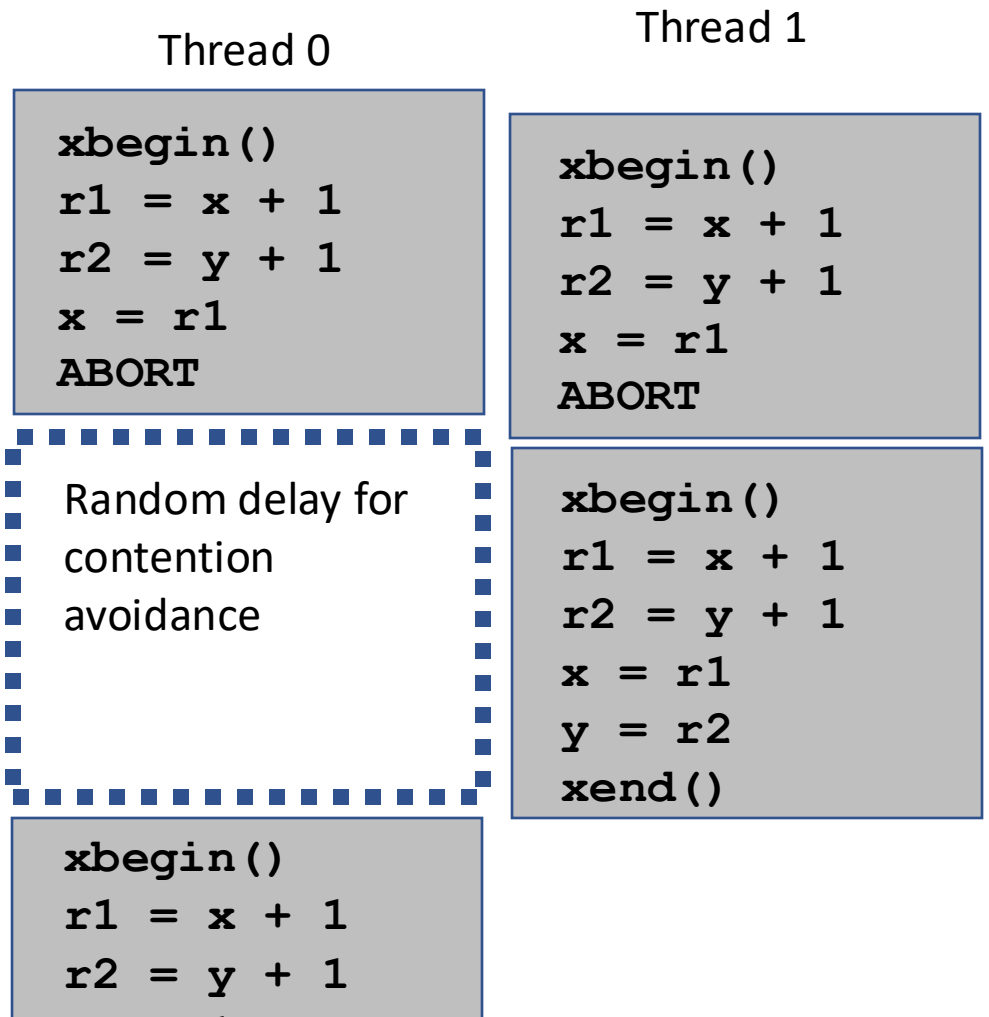
Lock-based Fallback Path

```
if(xbegin()==OK){
 rlk =
 read_spinlock(L)
 r1 = x + 1
 r2 = y + 1
 x = r1
 xend()
}else{
//fallback
 lock(L)
 r1 = x+1
 r2 = y+1
 x = r1
 y = r2
 unlock(L)
}
```

Add a fallback path & abort handling code
Fallback should use spinlocks, not TM. **Why?**
TM case needs to read spinlock lock word. **Why?**
In fallback, can do arbitrary code.
Can also retry TM version repeatedly before giving up and running fallback. Up to you the programmer what sequence to follow.

Precise Intel TSX syntax is available in the lab handout and tm.h in the lab release files.

What do we do if we have repeated aborts?



These threads are *contending for memory locations* causing repeated aborts.

How to deal with contention in a transactional memory system?

A Note About Lock-based Fallback Paths

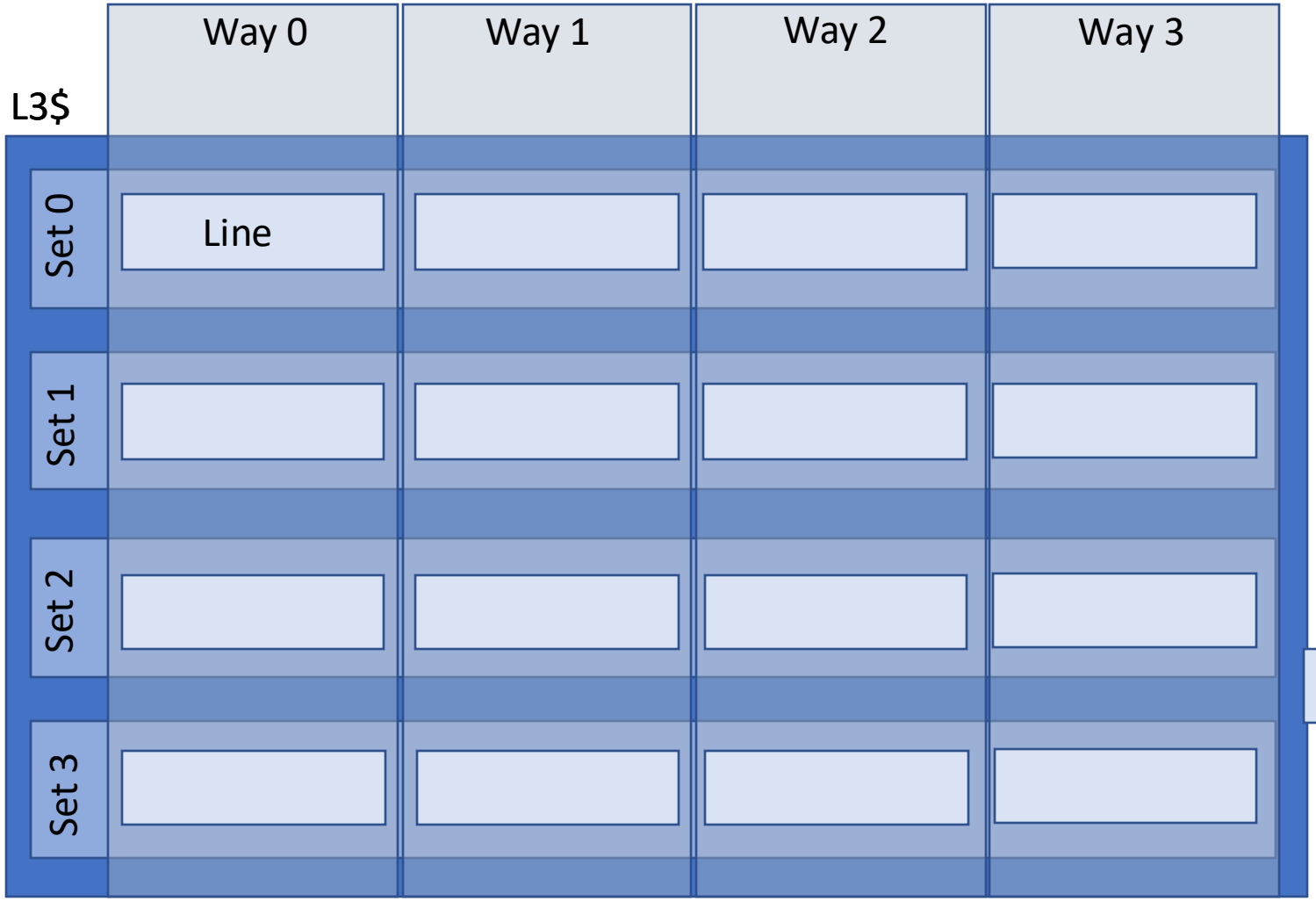
```
for(i = 0..MAX_TRIES){
 if(xbegin()){
  …; xend(); goto done;
 }//abort code here
}
//Fallback code here
lock(Lx); lock(Ly);
 r1 = x+1
 r2 = y+1
 x = r1
 y = r2
 unlock(Lx); unlock(Ly);
}

done:
 //continue
```

Run your transaction some number of times (MAX_TRIES)
If you commit once, skip past your fallback. Often use 'goto'…

Locks are tricky in code like this: which locks do you need to acquire? ***Often need to acquire them all before you make accesses associated with locks.***

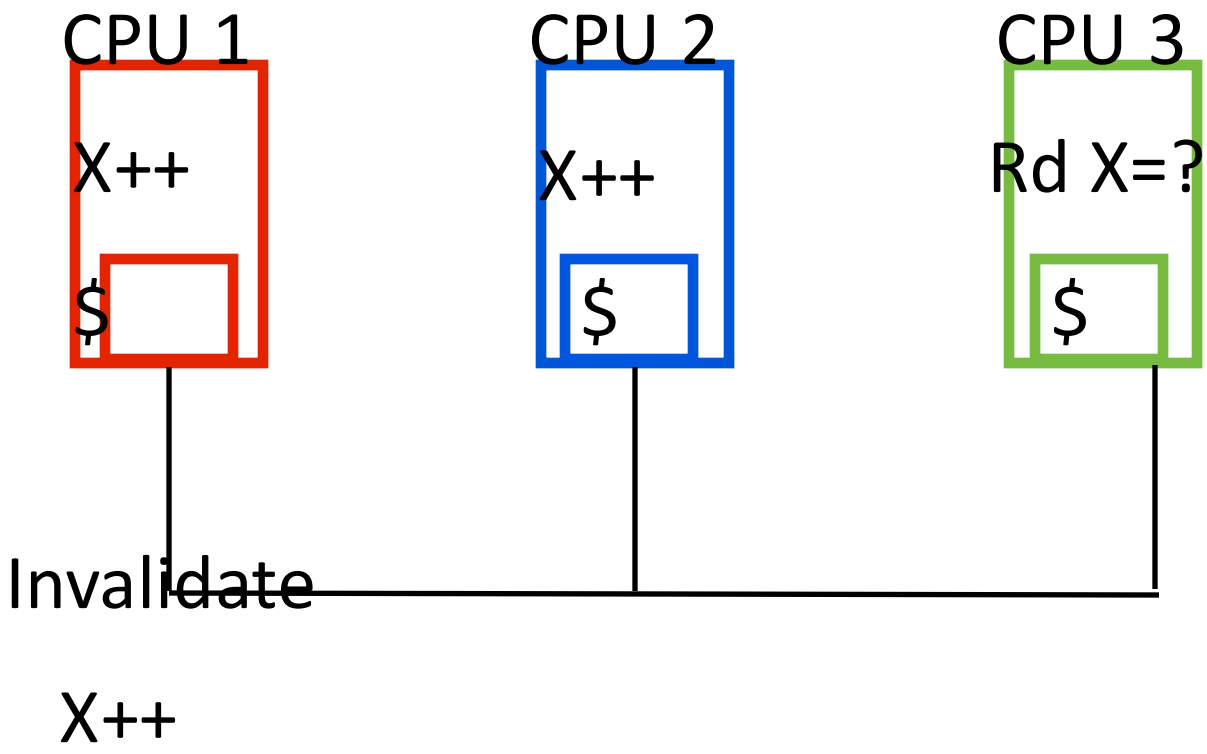
Implementation sketch of TM



Add TM bit to each cache block
Blocks accessed in transaction mark bit

All transactional state must fit in cache.

Tracking TM conflicts using coherence msgs



An incoming access request for a block with its TM bit set leads to a *conflict* and a transactional *abort*

Reasons a transaction might abort

- **Too many blocks with their TM bits set leaves no room for more TM blocks**
  - **Too many defined as “more blocks w/ TM bits set than blocks in a way”**
- **Conflict with another transaction *or non-transactional access***
  - **identified through incoming coherence traffic**
- **Explicit xabort() instruction when transactional code concludes transaction is not useful**
- **Other, unspecified, but arbitrary conditions left up to the microarchitects**
  - **I speculate that these are related to internal buffers of fixed capacity**

What did we just learn?

- Concurrency and parallelism, from the bottom to the top
- Coherence and consistency are both memory ordering principles
- Synchronization exists to spare you data-races and non-SC executions
- Transactional memory is a powerful sync primitive in many x86 CPUs