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

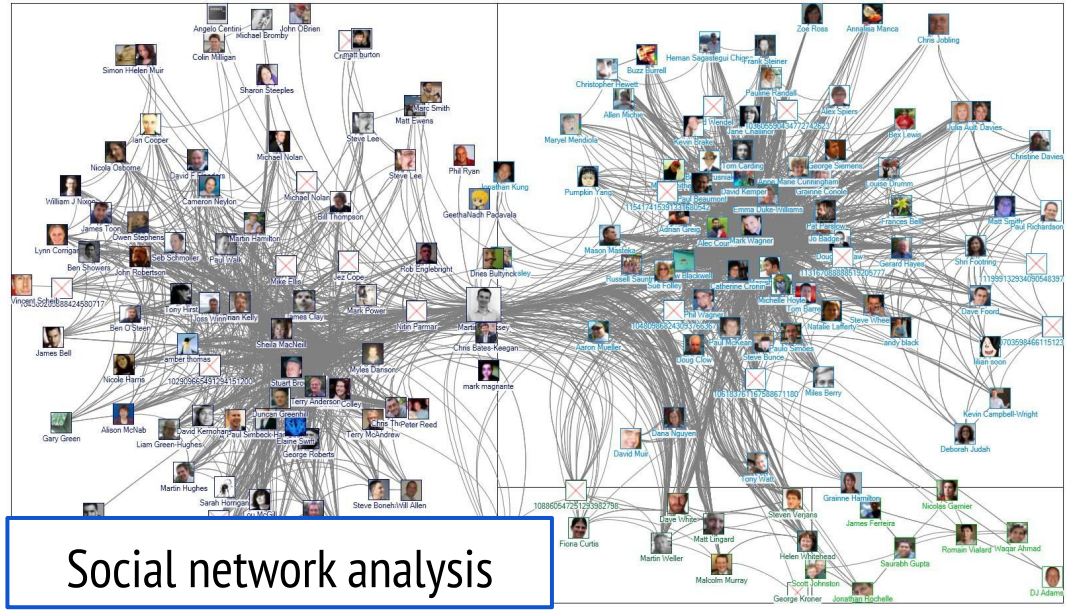
Fall 2021, Prof. Brandon Lucia

Recap: Sparse Problems

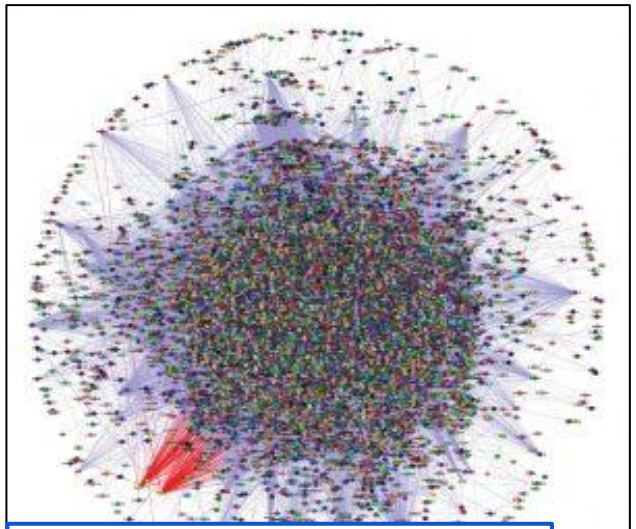
- What is a sparse problem? Why are they called “sparse”?
- What makes sparse problems hard?
- Roofline performance modeling
- Hardware and software strategies for optimizing sparse problems

Graph Processing Problems are Sparse Problems

Path Planning



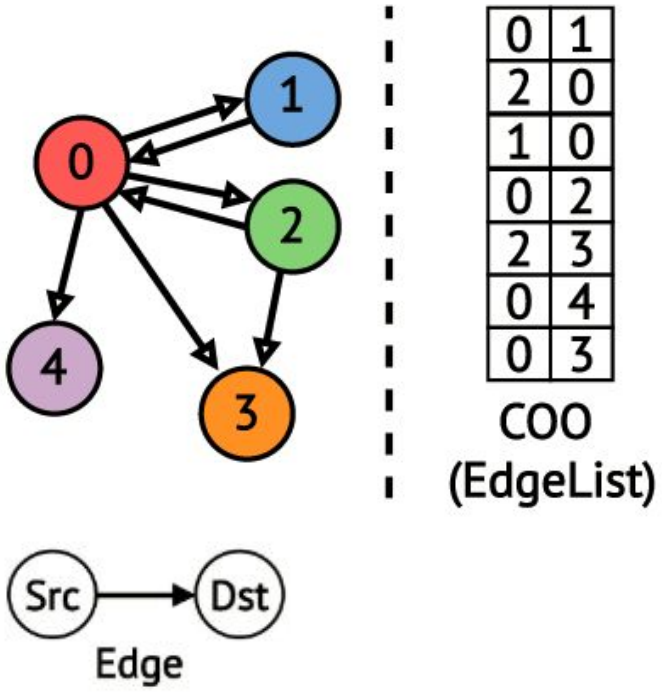
Social network analysis



Protein-Protein Interaction

The canonical examples of sparse problems are graph processing applications.

What does a graph processing program look like?



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

COO
(EdgeList)

```
for e in EL:
  dstData[e.dst] =
     f(srcData[e.src],dstData[e.dst])
```

dstData

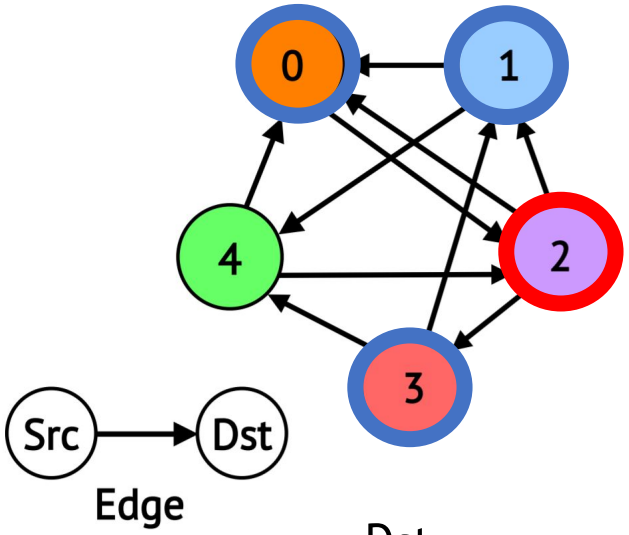
srcData

stores vertex property information

if srcData == dstData, updating in-place;

often “swap” srcData & dstData from 1 iteration to the next iteration

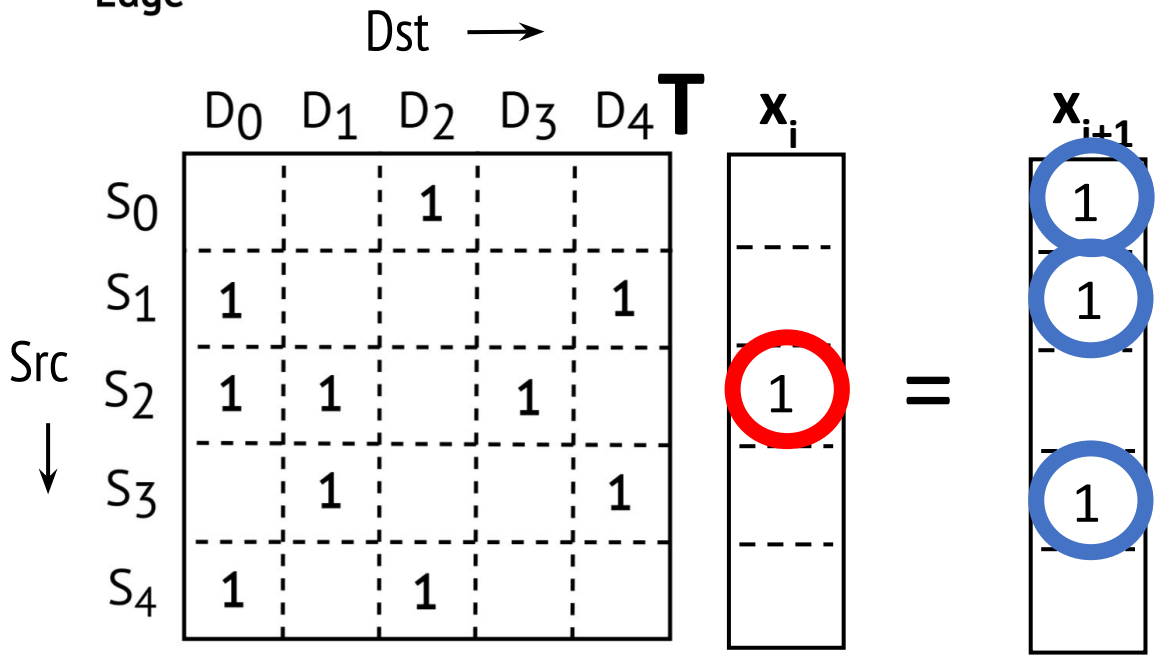
How do graph applications correspond to linear algebra?

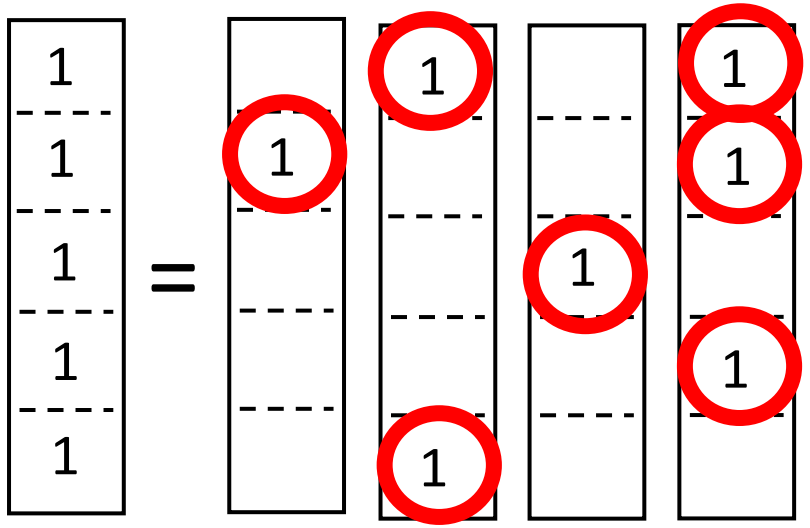


Turns out that other graph applications also correspond to roughly this formulation if you change the operations you use (min/+ instead of +/*) or consider weighted edges

$$\mathbf{A}^{\mathbf{T}}x_i = x_{i+1}$$

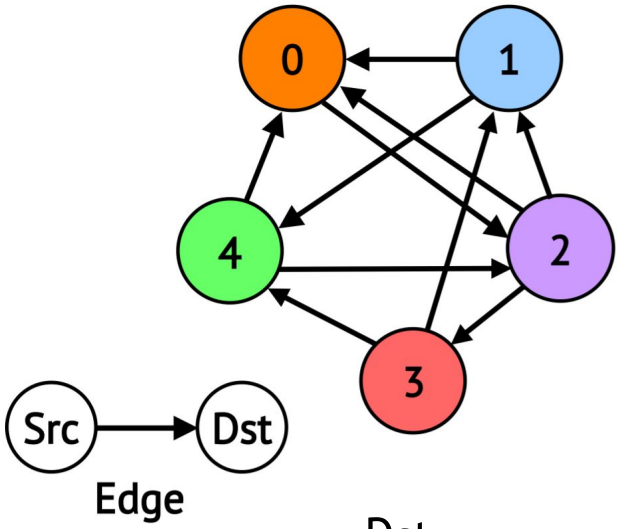
SSSP, BFS, PageRank, Connected-Components, Betweenness-Centrality, triangle counting… BFS is a representative sparse problem.

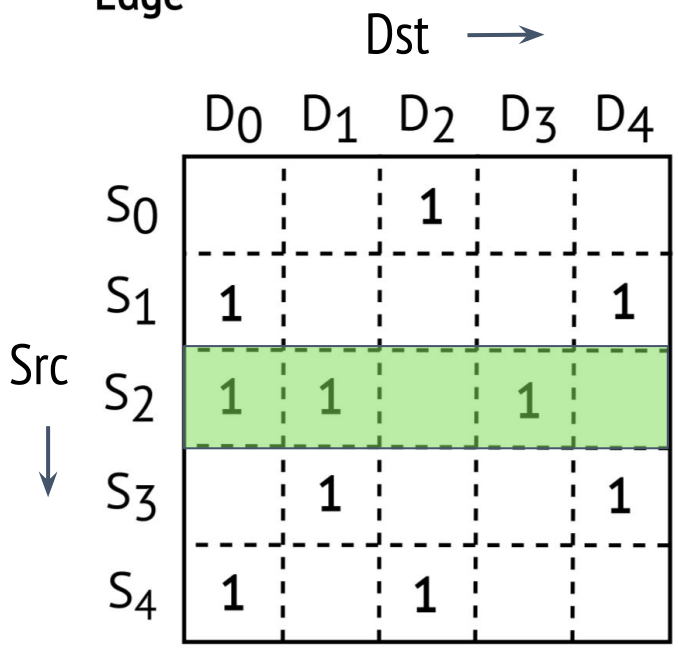




Search done when no new vertices added (or all visited)

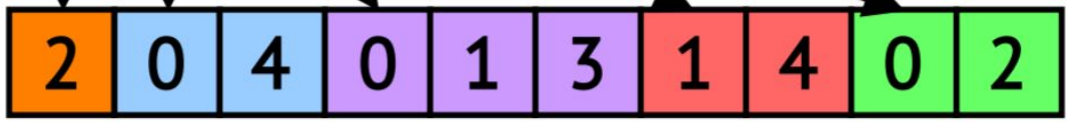
Compressed Sparse Data Structures for Feasible Memory Size





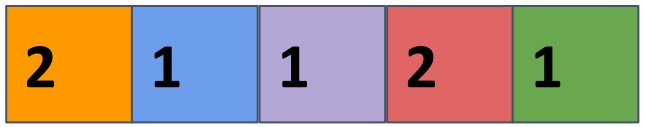
Offsets Array (OA)

Neighbors Array (NA)



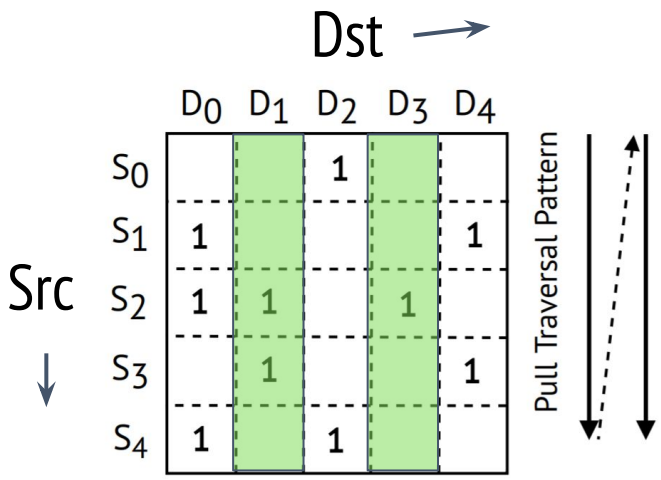
Compressed Sparse Row (CSR)
Outgoing Neighbors

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

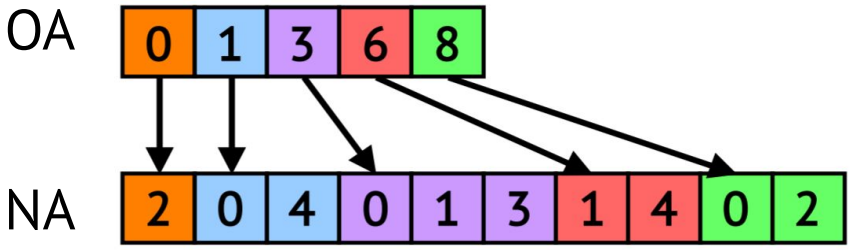
Compressed Representations ⇒ Irregular Memory Accesses



Push (CSR Traversal)

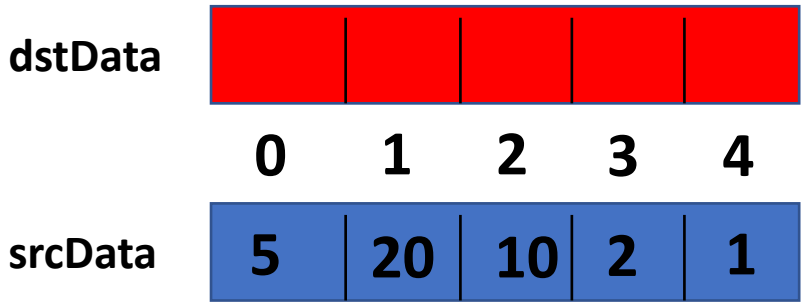
```
for src in G:
   for dst in out_neighs(src):
         dstData[dst] += srcData[src]
```

Push traversal performs *irregular write operations* that lack locality



CSR

i.e., x_{i+1}

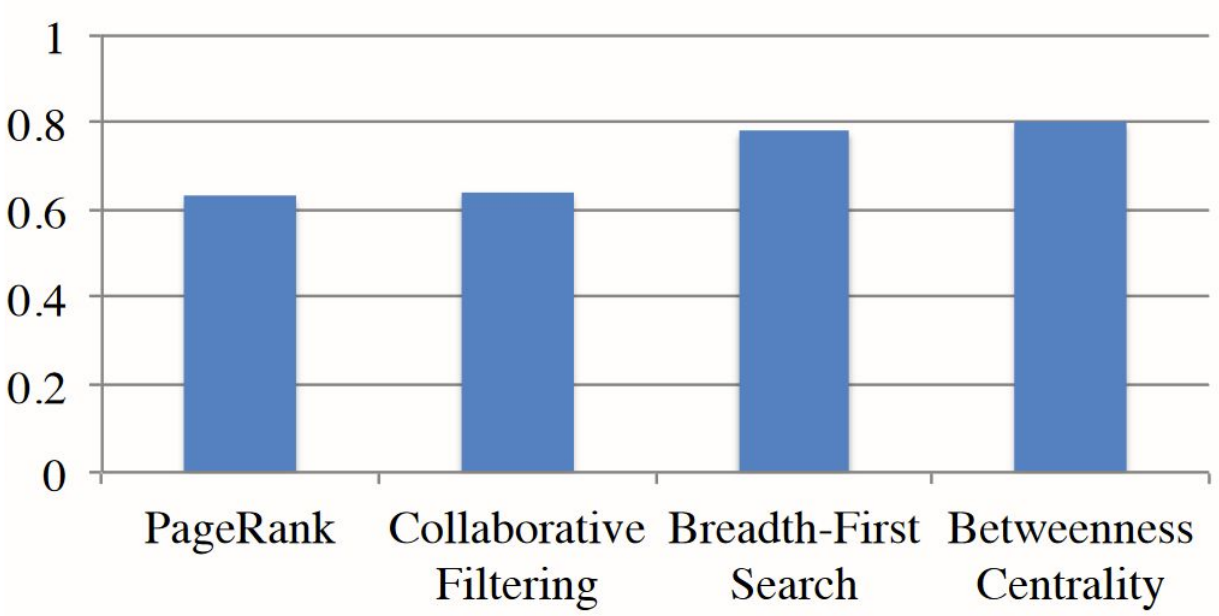


e.g., current rank of page I,
e.g., current shortest path from source vertex

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

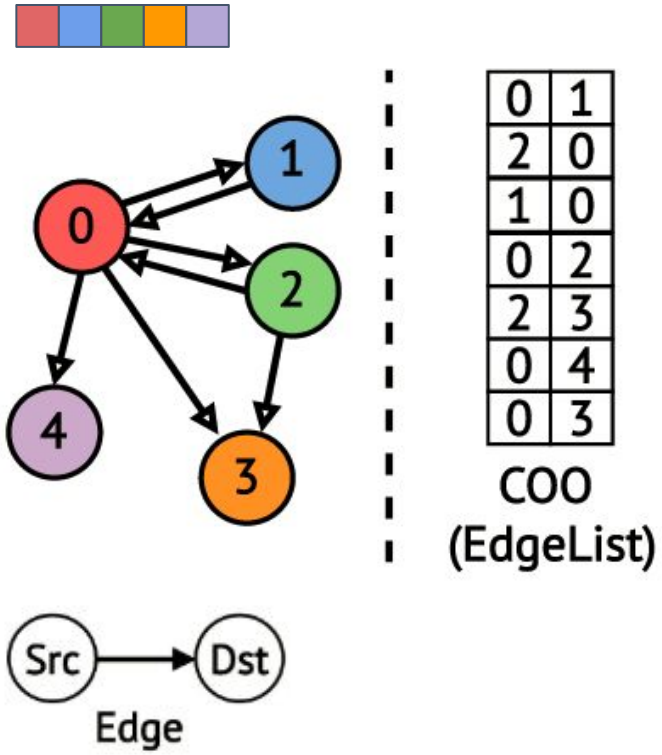
Cycles stalled on DRAM / Total Cycles



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!

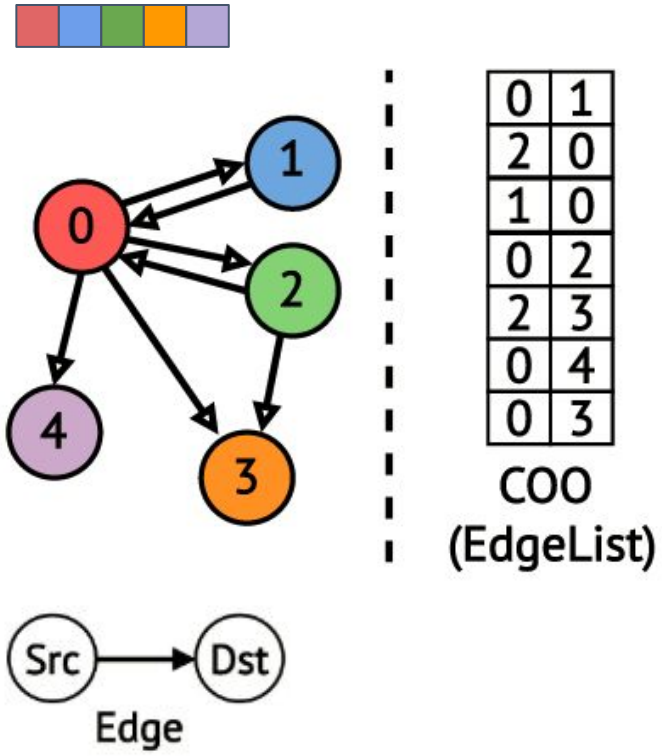



```
for e in EL:
  neigh_count[e.dst]++; /*e.src*/
```

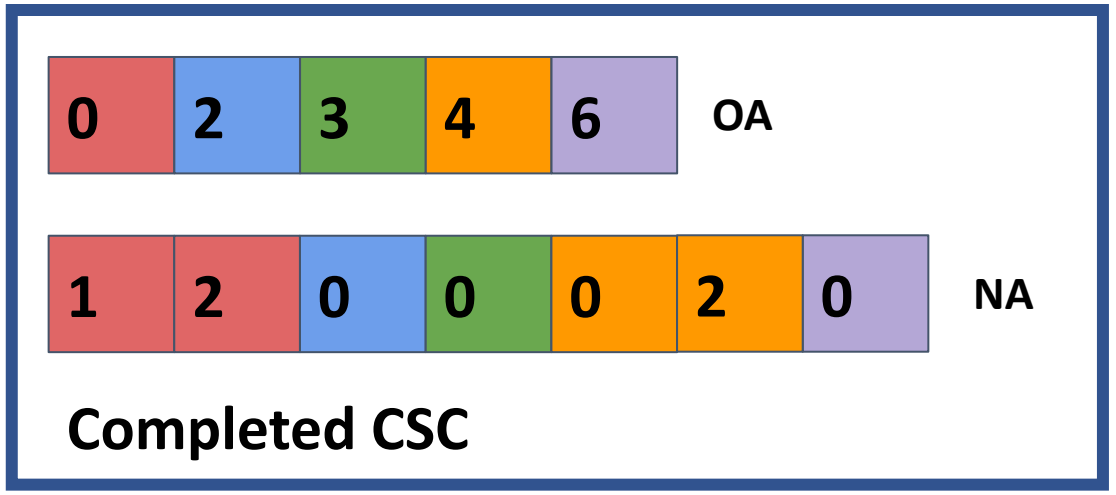
neigh_count

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

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



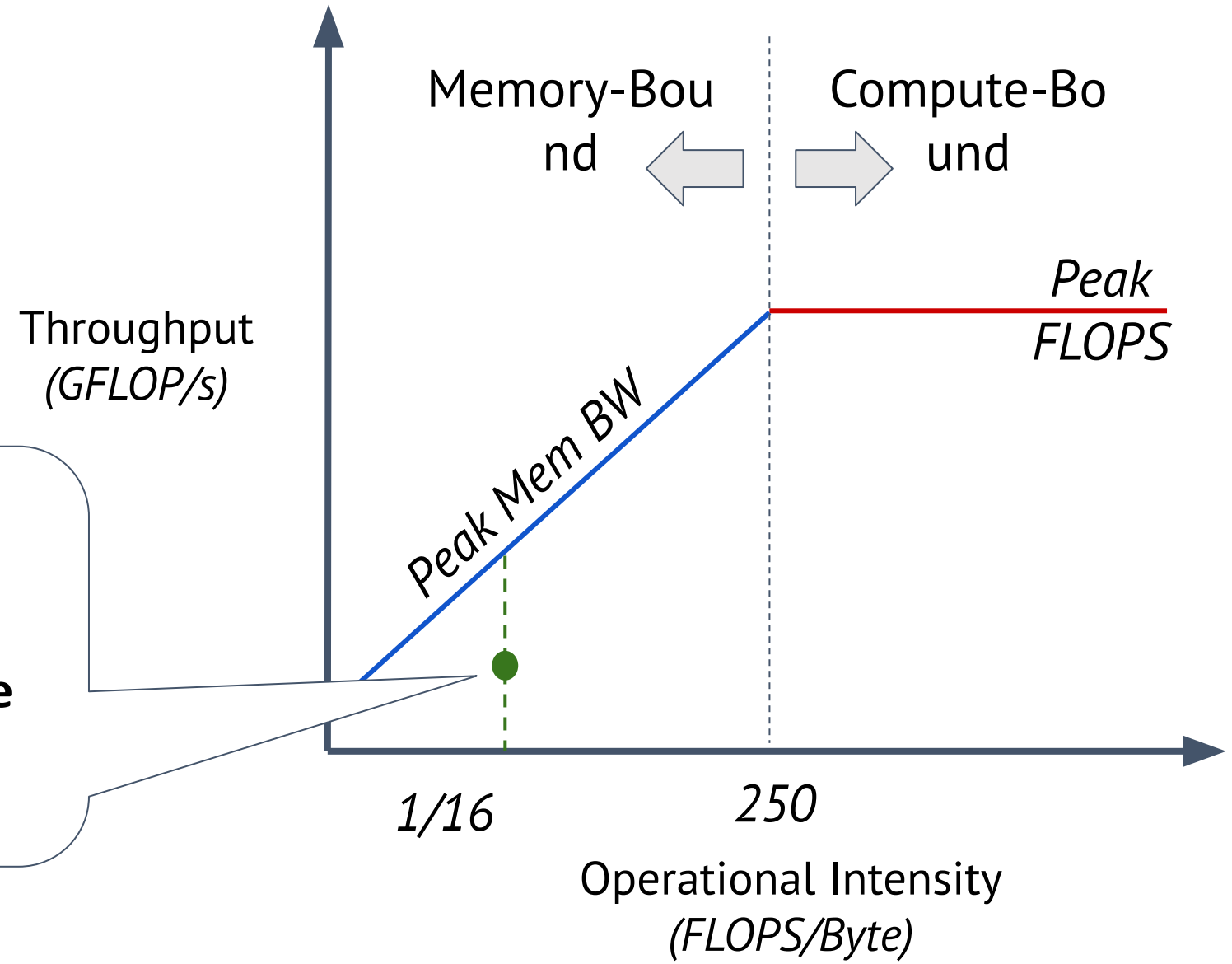

```
for e in EL:
  NA[ OA[e.src]++ ] = e.dst
```



Updates to NA based on EL order & OA[e.src]

NA[OA[e.src]++] = e.dst

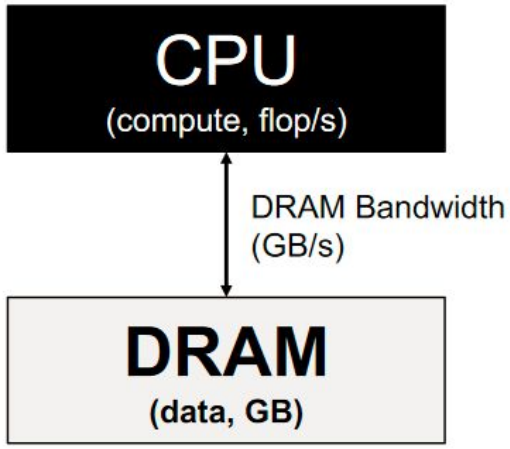
Graph Applications are Memory-Bound

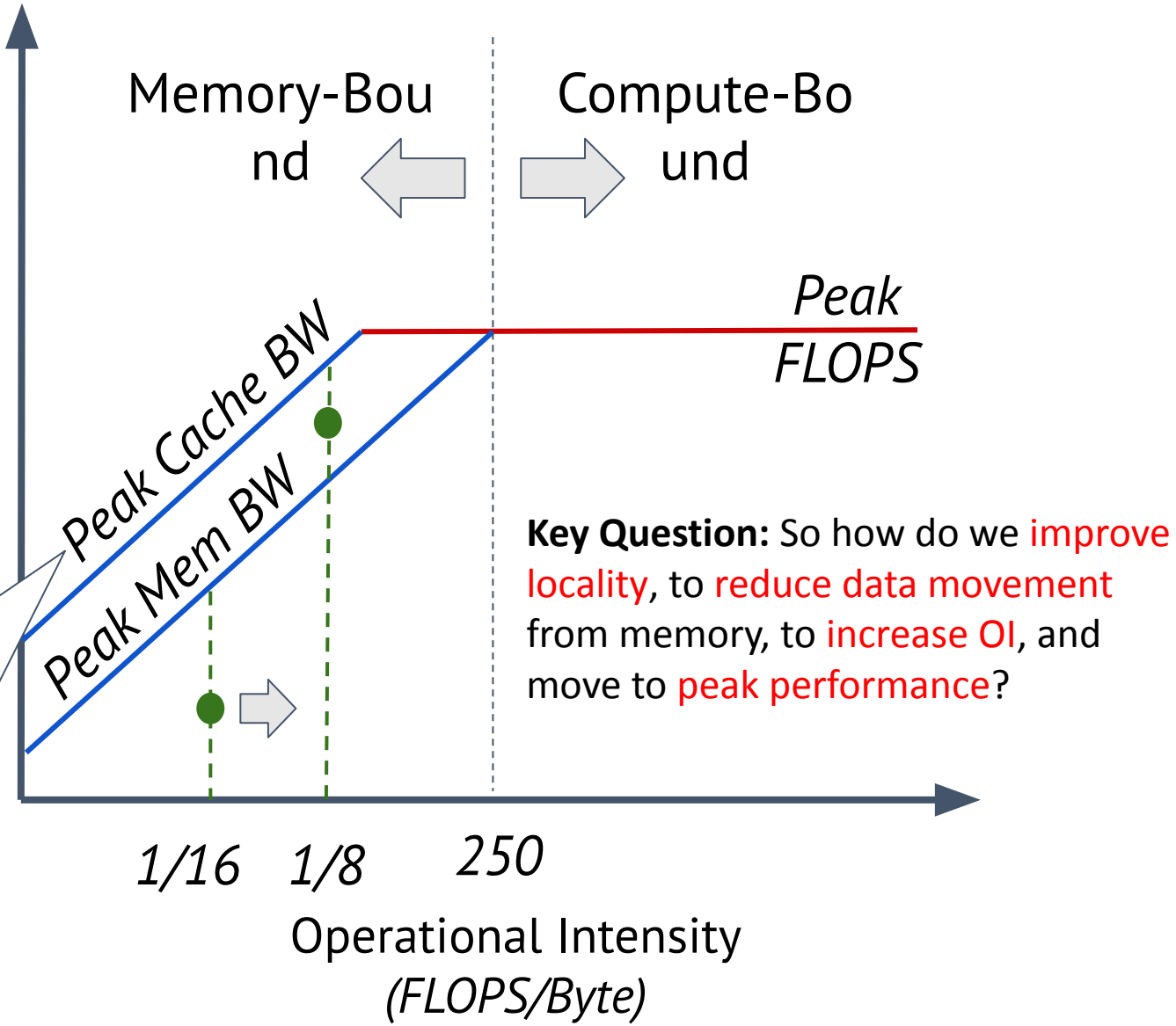


DRAM BW utilization in graph apps is ~50%

Why would we have spare BW capacity to go to memory and not use it?

Improving Operational Intensity (OI) by Improving Locality





Locality wins: If we can operate out of cache, higher ceiling & more leftward ridge point.

Why is cache BW > DRAM BW?
Smaller SRAM caches much faster.

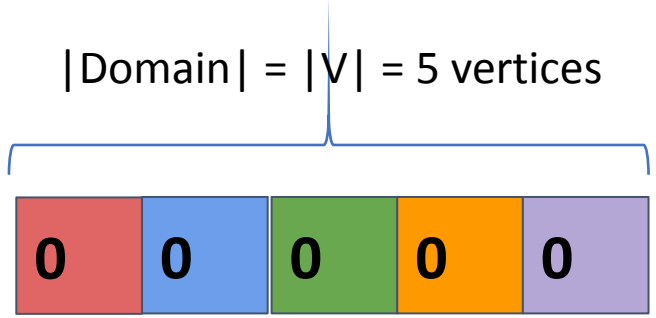
Key Question: So how do we improve locality, to reduce data movement from memory, to increase OI, and move to peak performance?

Propagation Blocking: Optimizing Sparse Irregular Writes to Improve Cache Locality

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

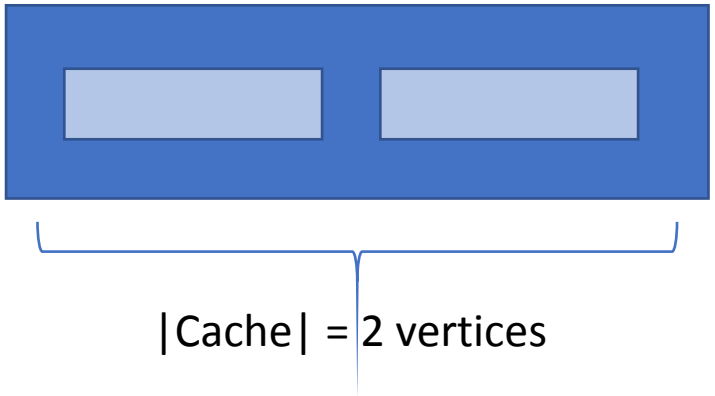
0	1
2	0
1	0
0	2
2	3
0	4
0	3

COO
(EdgeList)



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

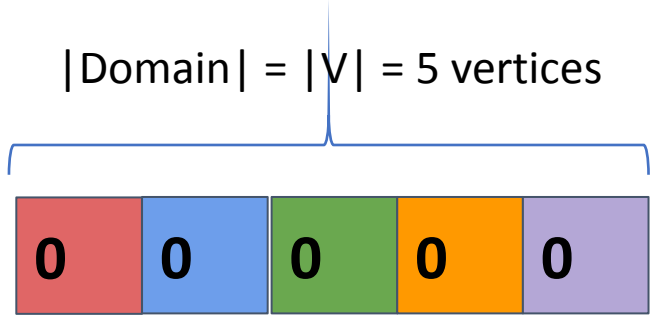
Bad for the cache: the size of the *domain* of vertex data array entries is |V|, but the cache holds only |C| << |V| entries



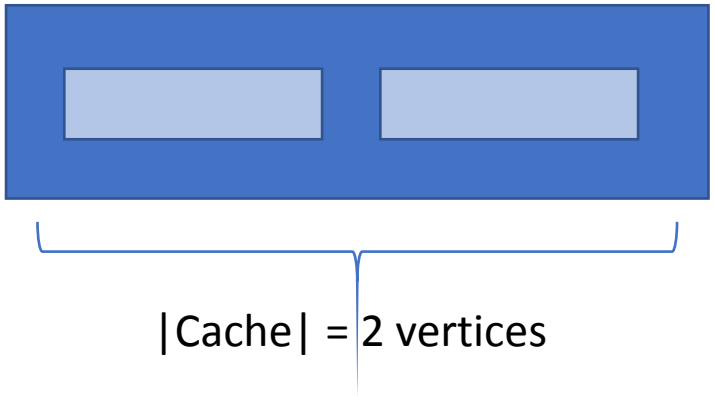
Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

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

COO
(EdgeList)



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



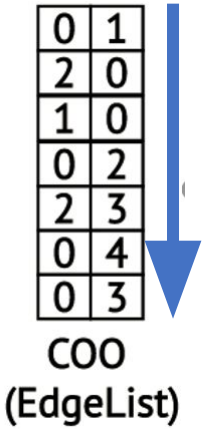
Recall: irregular accesses into vertex data array based on e.dst *which are essentially random*

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

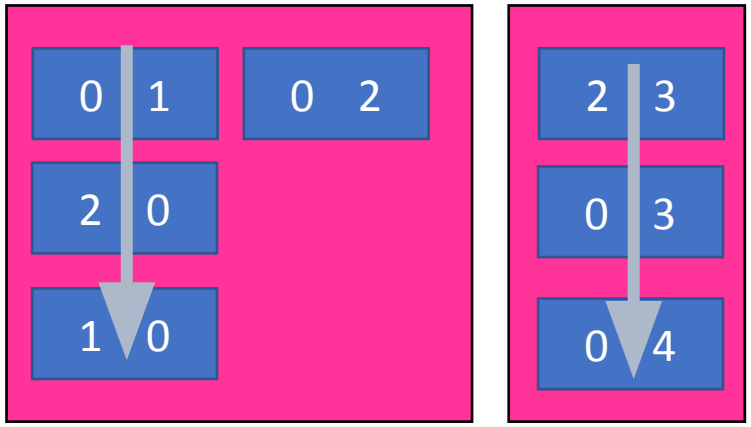
Propagation Blocking: Performance Analysis

Traverse the edge list twice instead of once

Binning



Bin Read

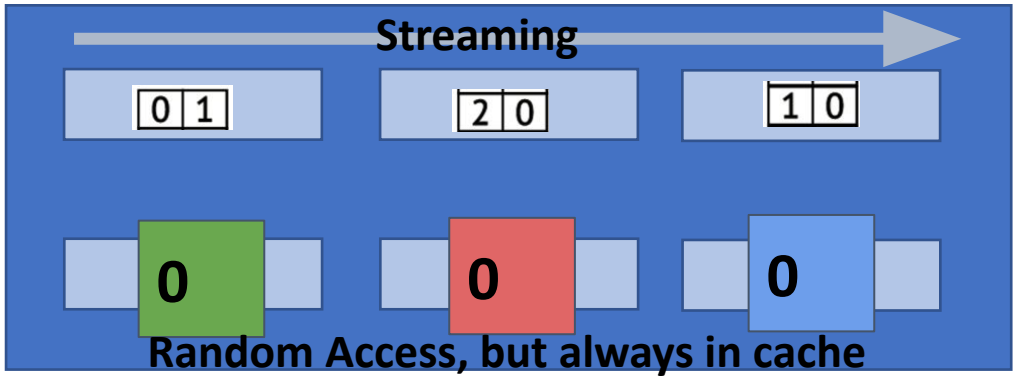


Bin 1:
dst 0-2

Bin 1:
dst 3-5

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

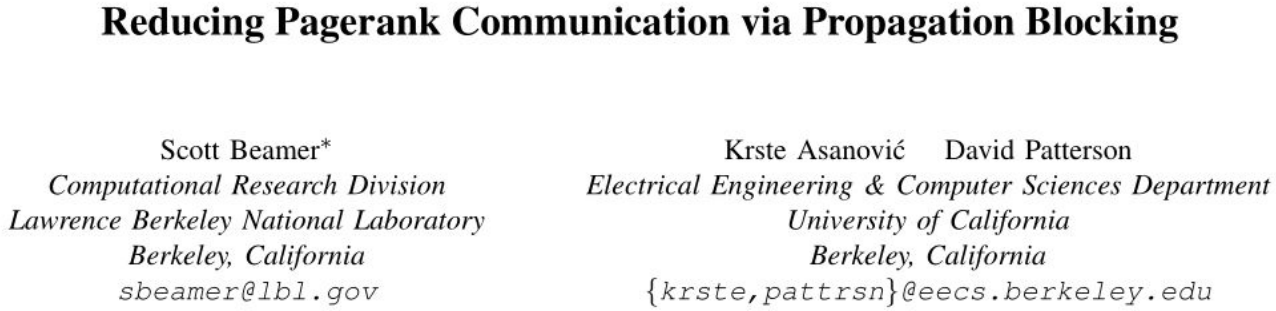
Propagation Blocking

```
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]++
    }
  }

}
```



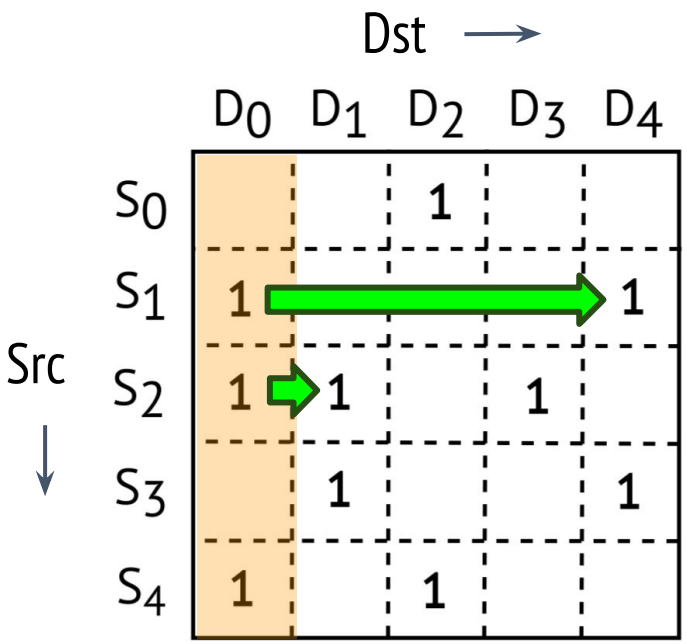
Reducing Pagerank Communication via Propagation Blocking

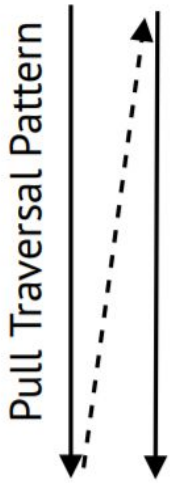
Scott Beamer*
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California
sbeamer@lbl.gov

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)

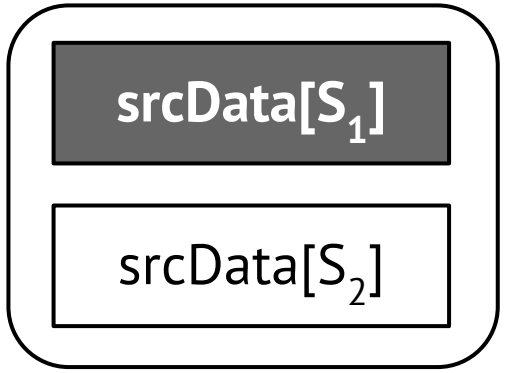
Using The Graph's Transpose For Optimal Replacement





Pull Execution (*CSC Traversal*)

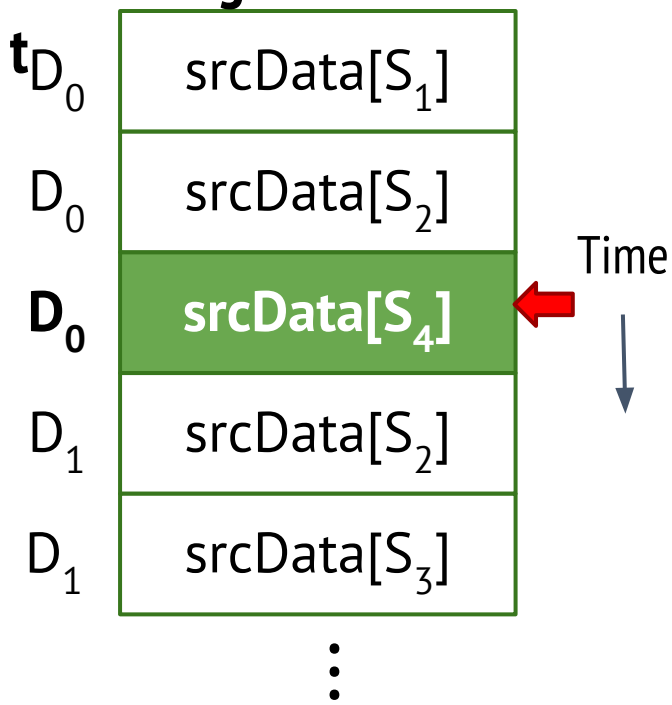
```
for dst in G:
   for src in in_neighs(dst):
      dstData[dst] += srcData[src]
```



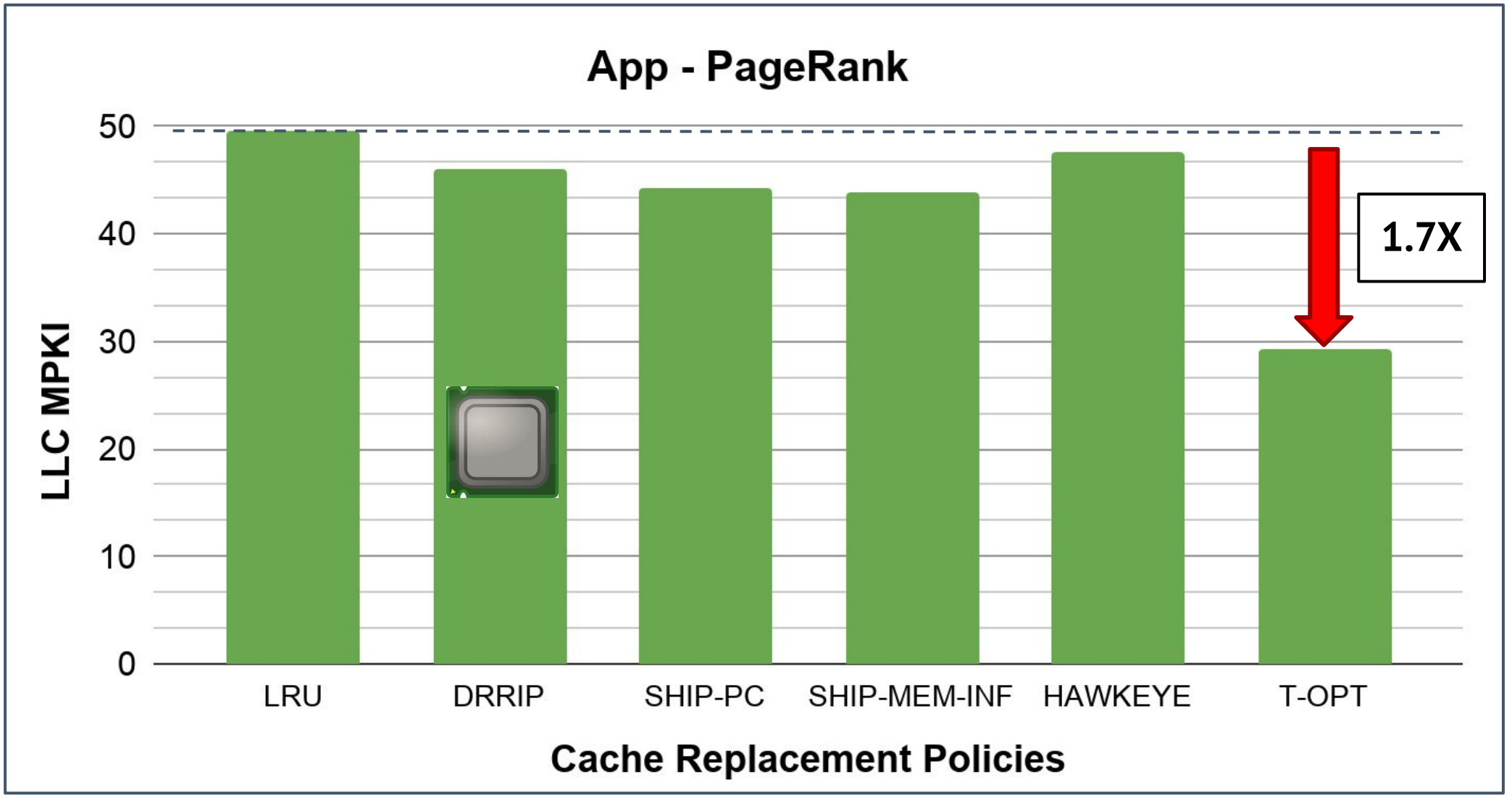
2-way Set-Associative

Which line should we evict?:

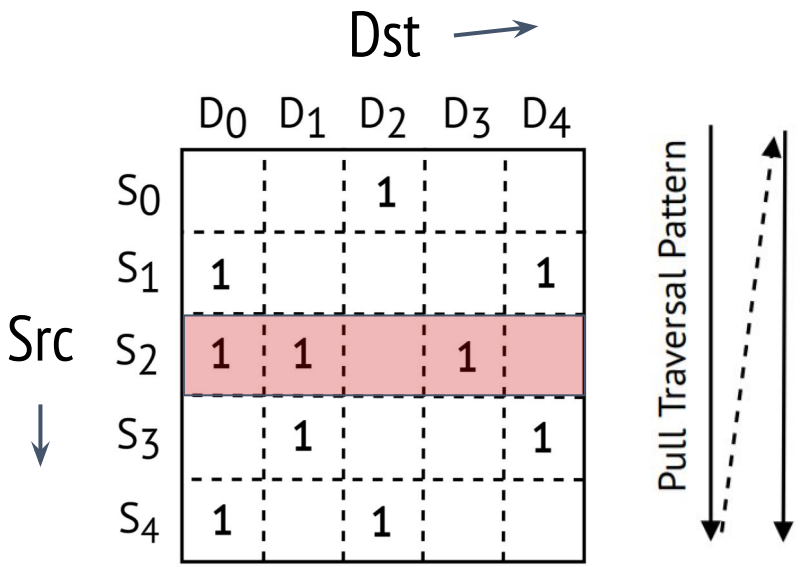
- srcData[S_1] **(nextRef @ D_4)** ✔
- srcData[S_2] **(nextRef @ D_1)**



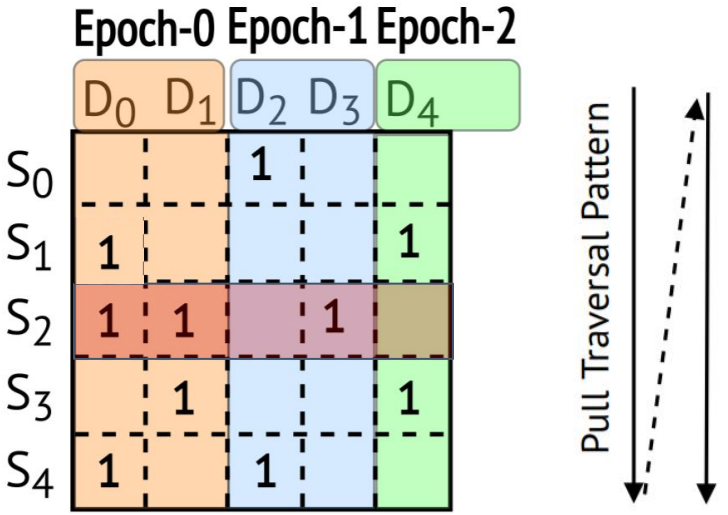
Transpose-based OPT (T-OPT) Provides Large Gains

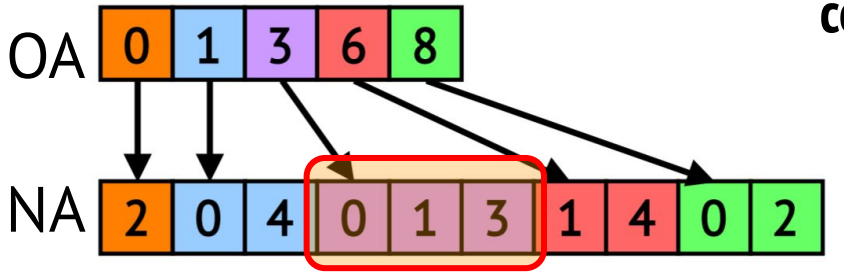


Main Technique: Use Quantization To Compress The Transpose



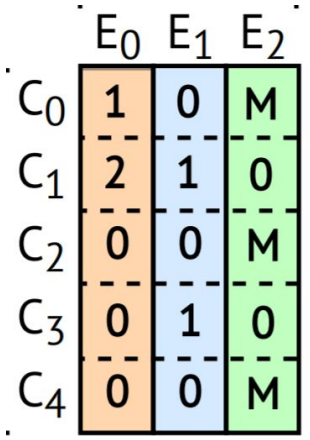
Divide execution into coarse-grained epochs





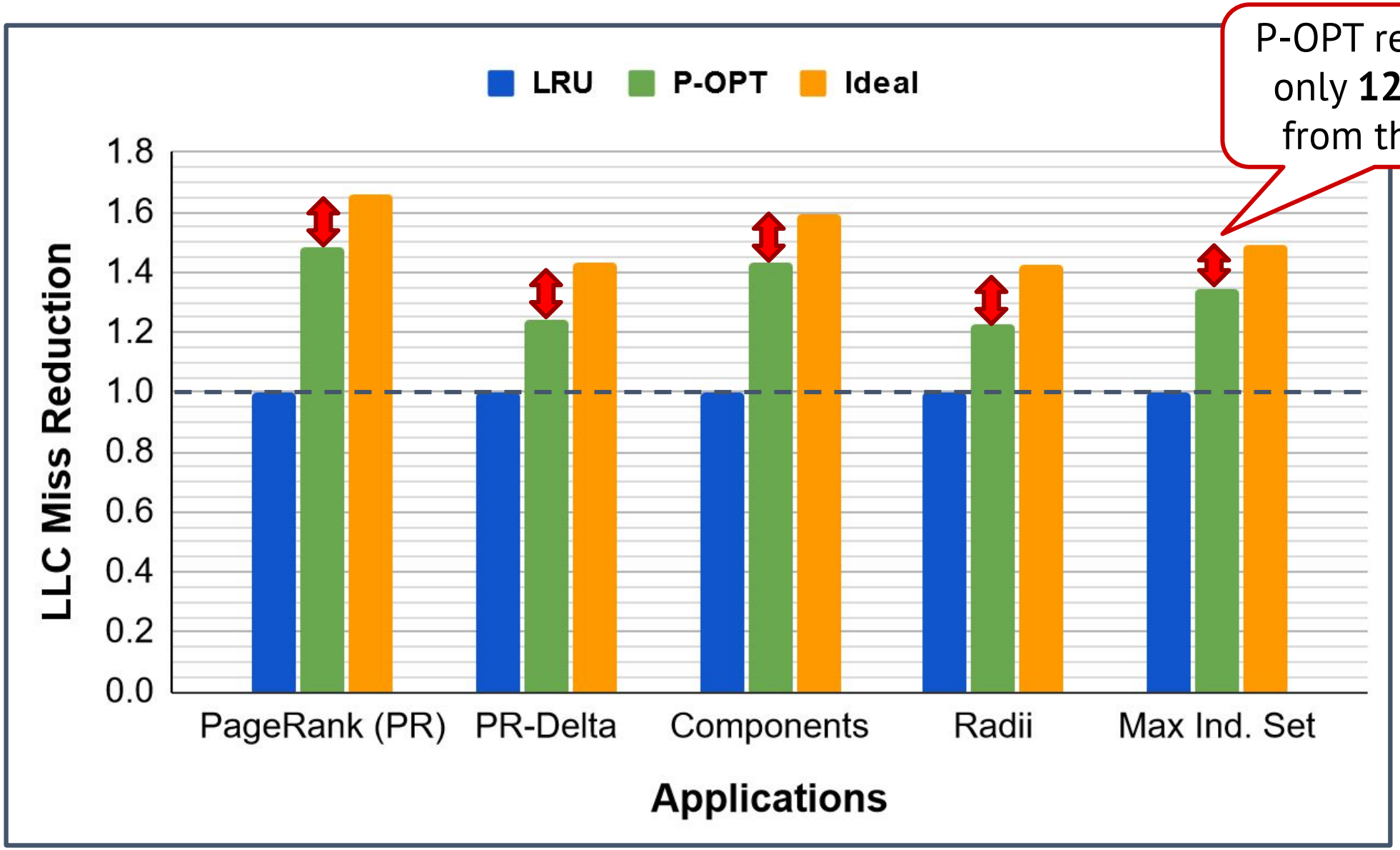
CSR
(Transpose)

Quantization enables compression of transpose data



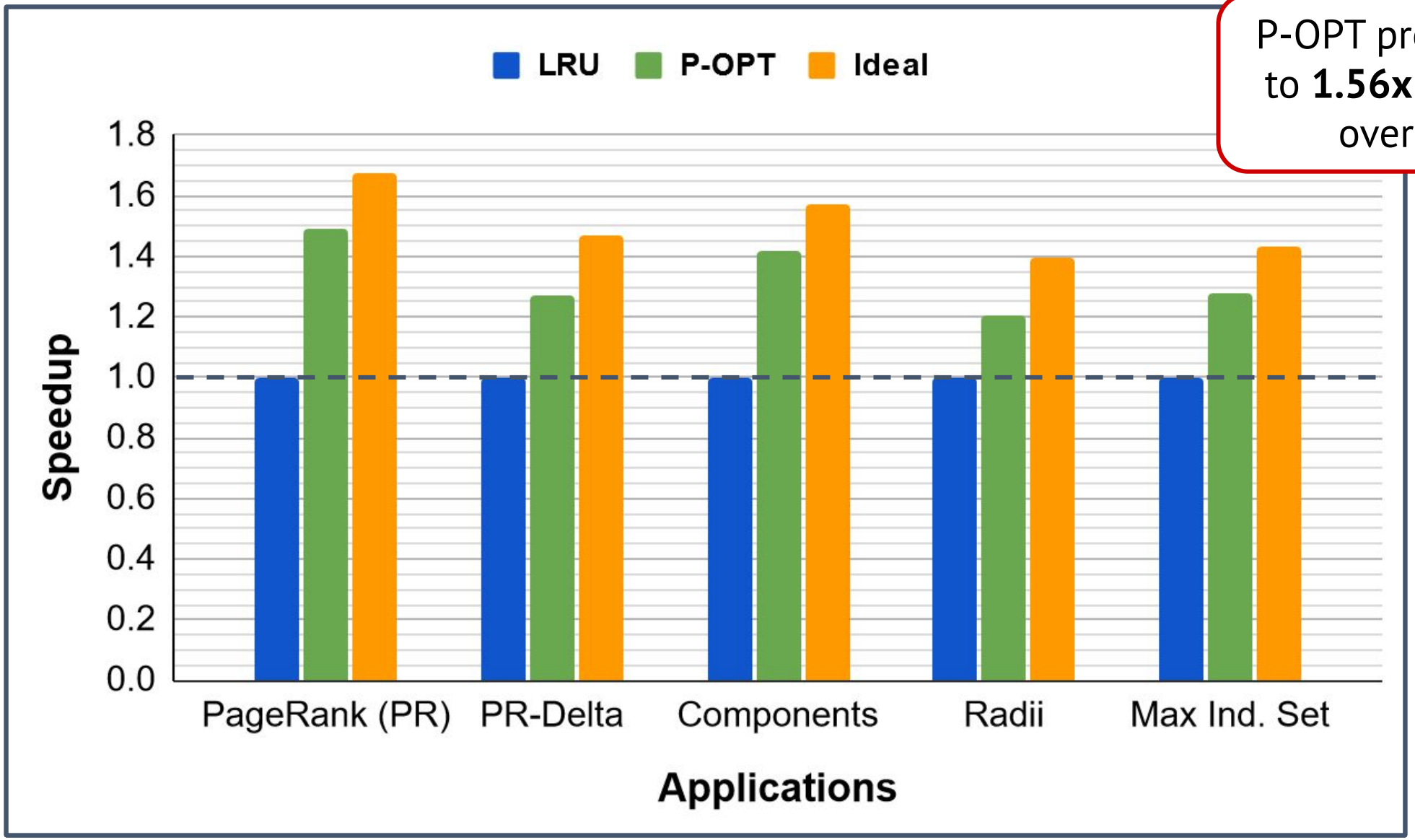
Rereference Matrix
(Quantized Transpose)

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



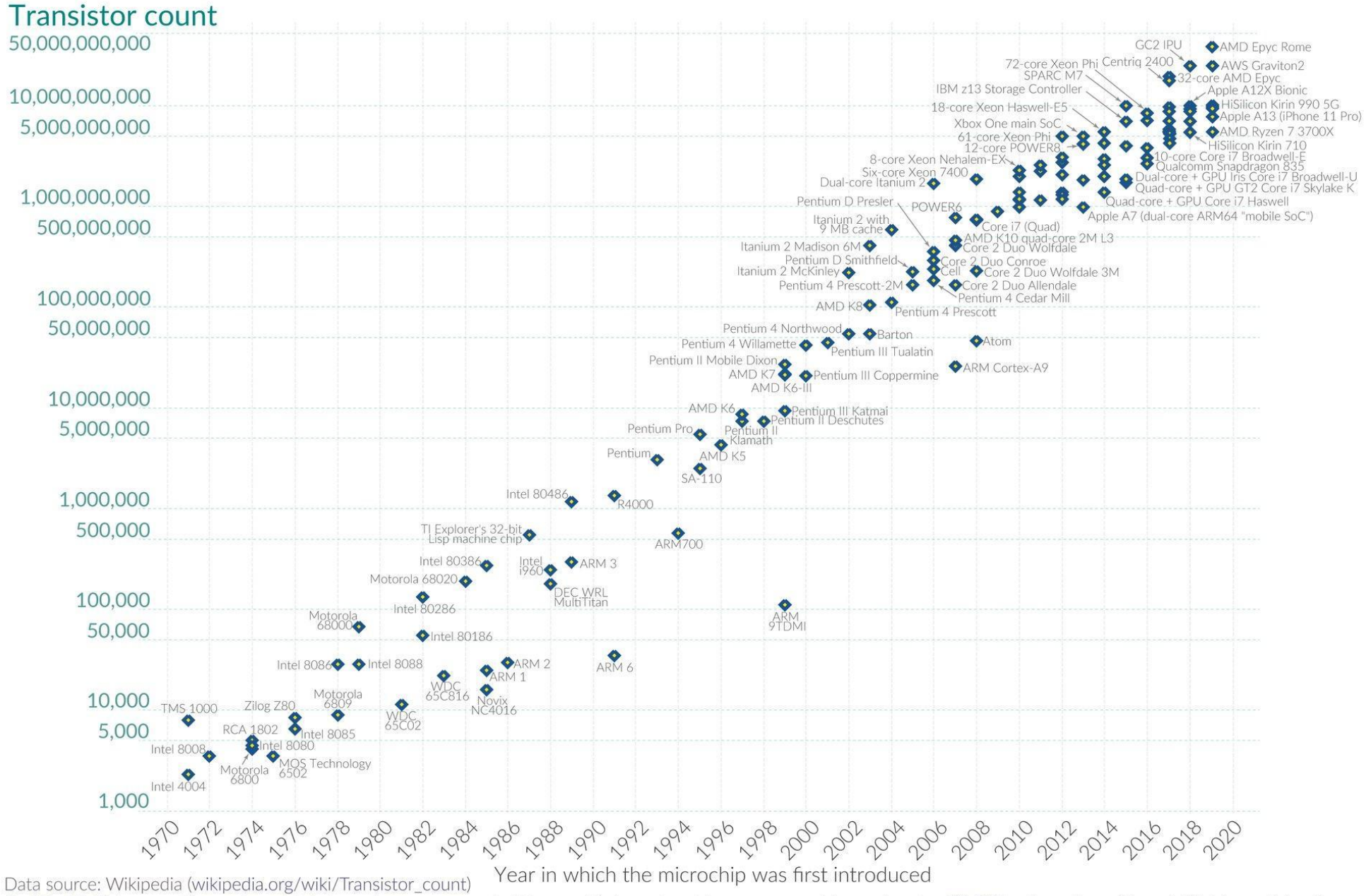
Today: Parallel Computer Architectures

- 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

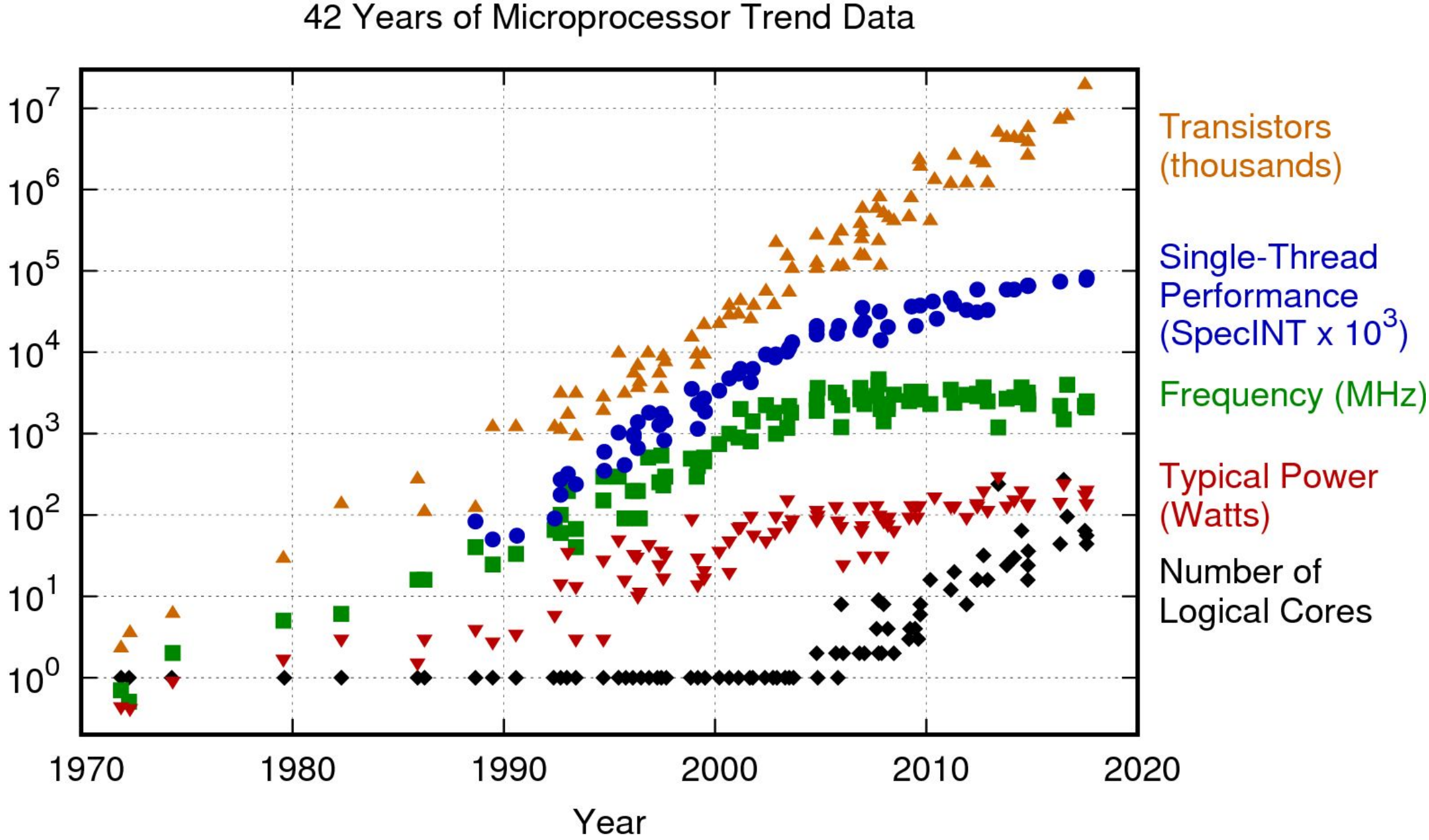
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World in Data



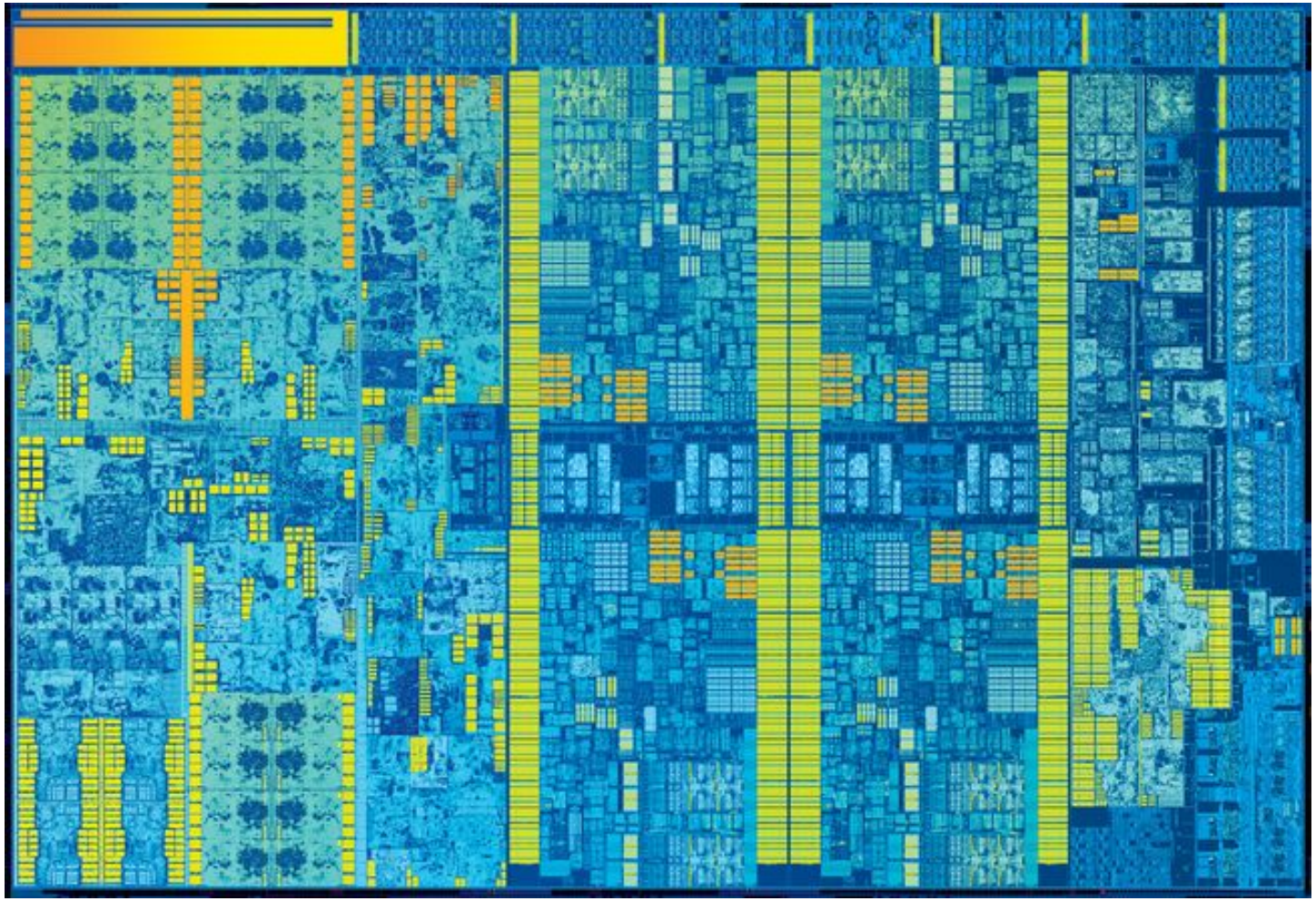
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

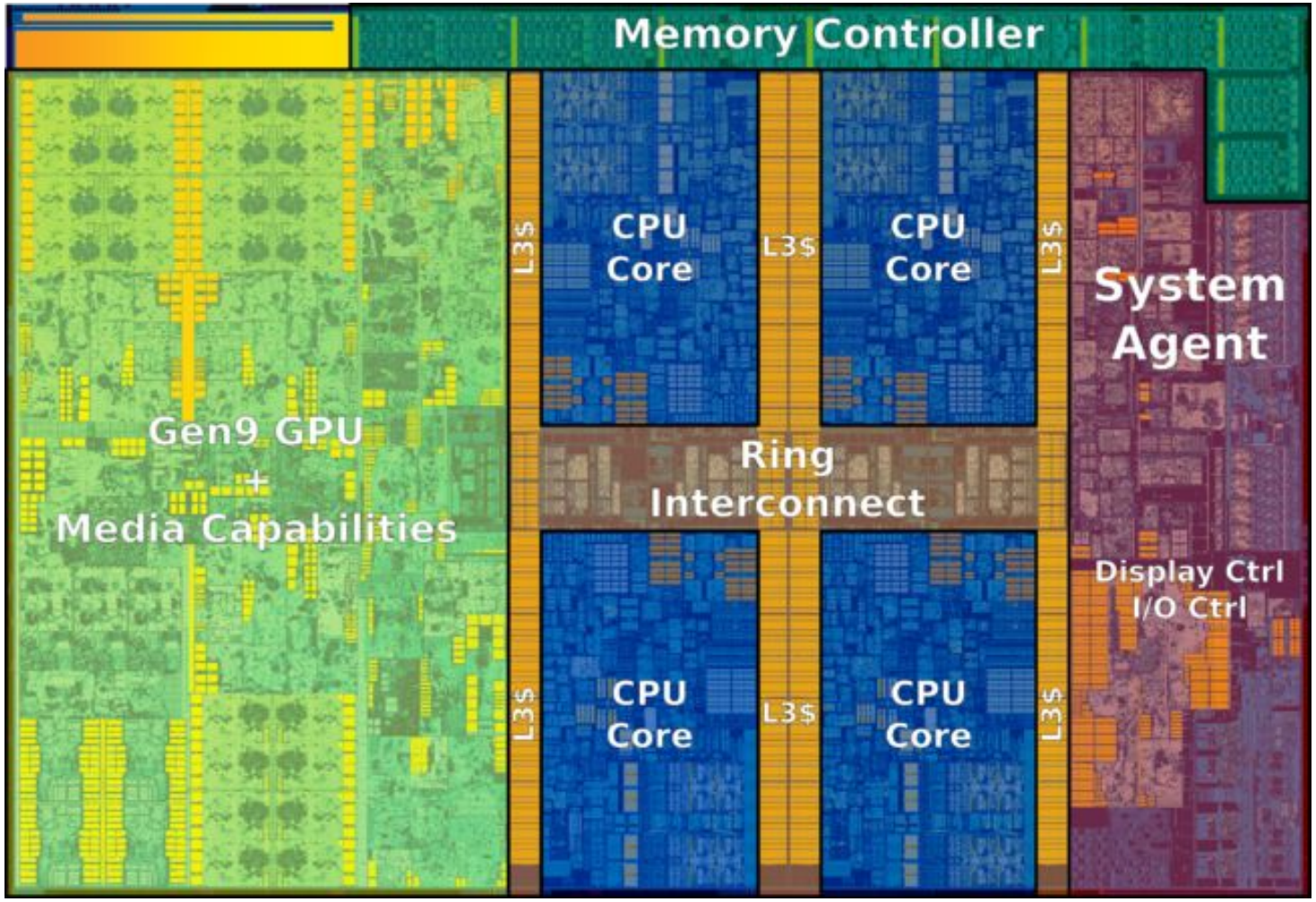


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

- 14 nm process
- 11 metal layers
- ~1,750,000,000 transistors
- ~9.19 mm x ~11.08 mm
- ~101.83 mm² die size
- 4 CPU cores + 24 GPU EUs

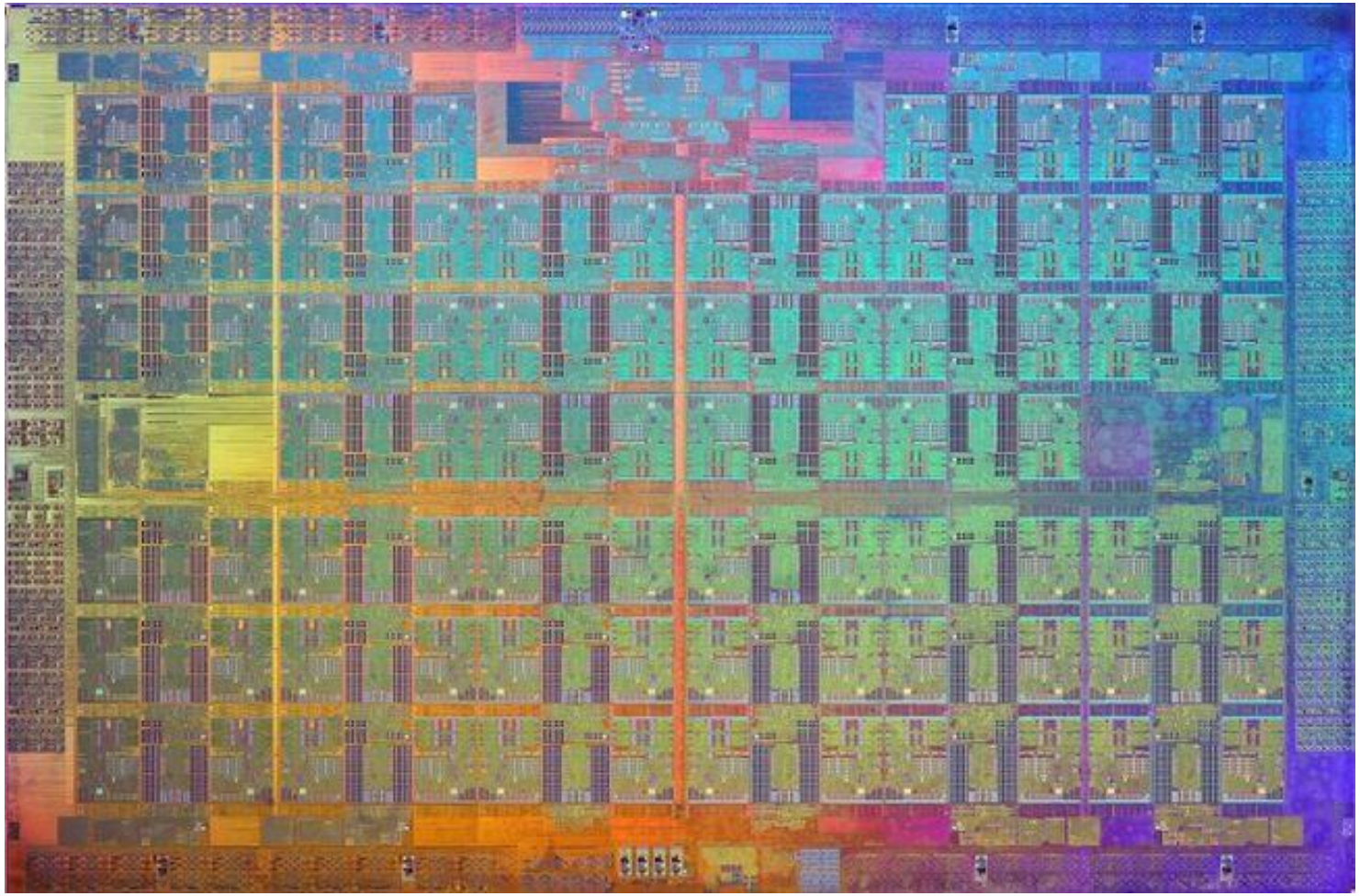


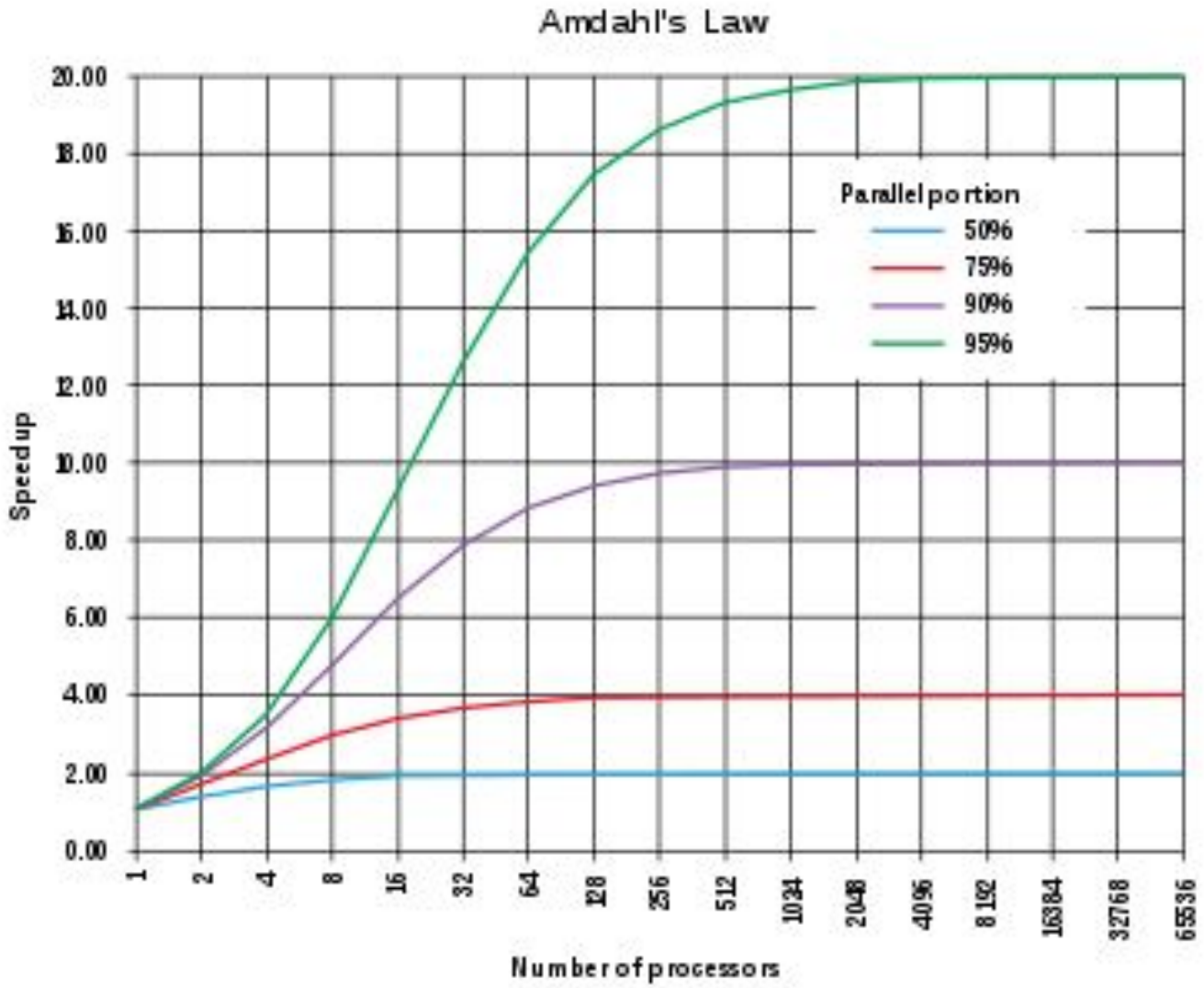
- 14 nm process
- 11 metal layers
- ~1,750,000,000 transistors
- ~9.19 mm x ~11.08 mm
- ~101.83 mm² die size
- 4 CPU cores + 24 GPU EUs



Shared memory multi-threading

- 14 nm process
- 682.6 mm² die size
- 76 CPU cores
- 7,100,000,000 transistors





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

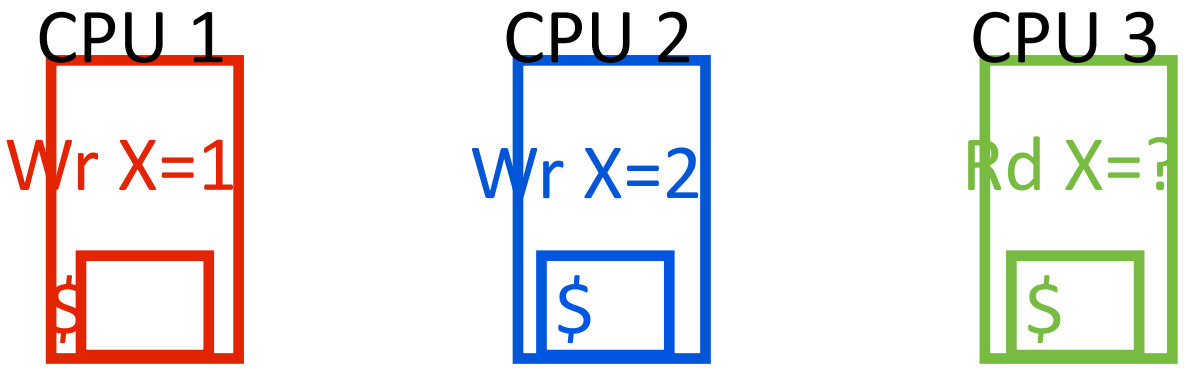
Multi-core parallelism was the primary way to keep performance scaling alive once single-thread performance hit the wall

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

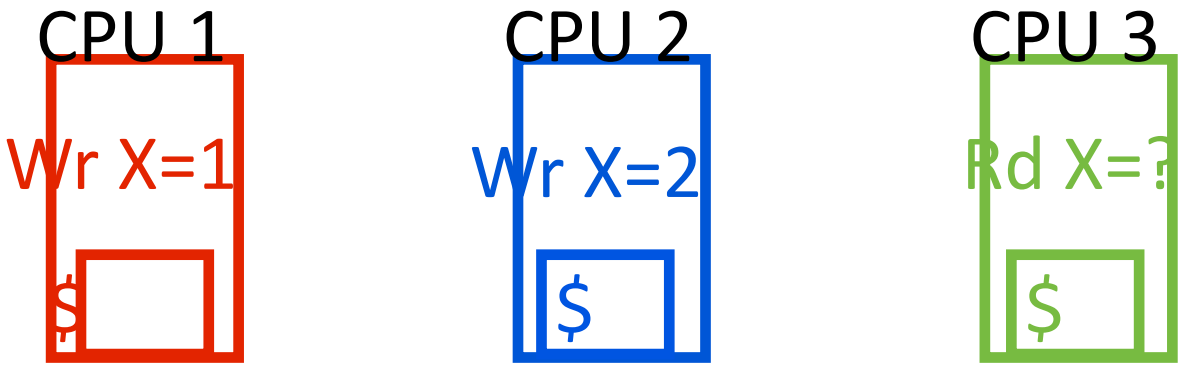
"Coherence seeks to **make the caches of a shared-memory system as functionally invisible as the caches in a single-core system**. Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores."

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)



Wr X=1

Wr X=2

Rd X=**2**

Wr X=2

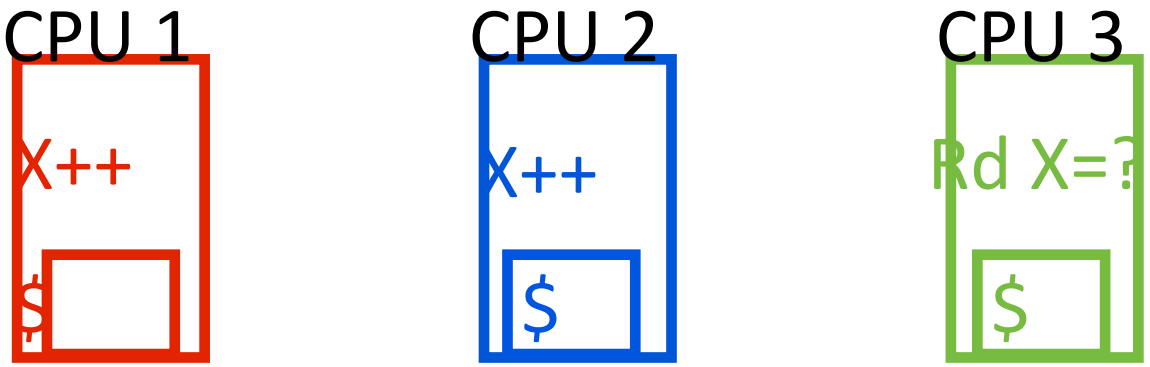
Wr X=1

Rd X=**1**

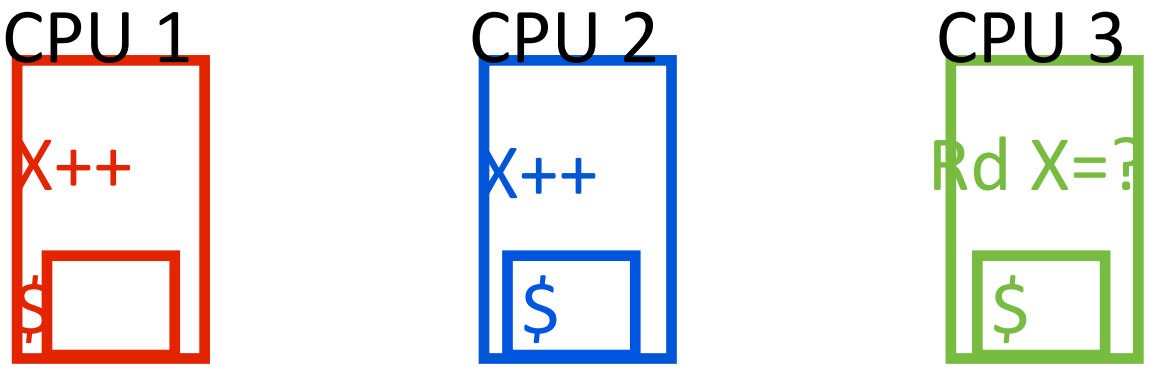
Rd X=**0**

Wr X=1

Wr X=2



What about this example?
(X initially 0)



X++

X++

X++

X++

X++

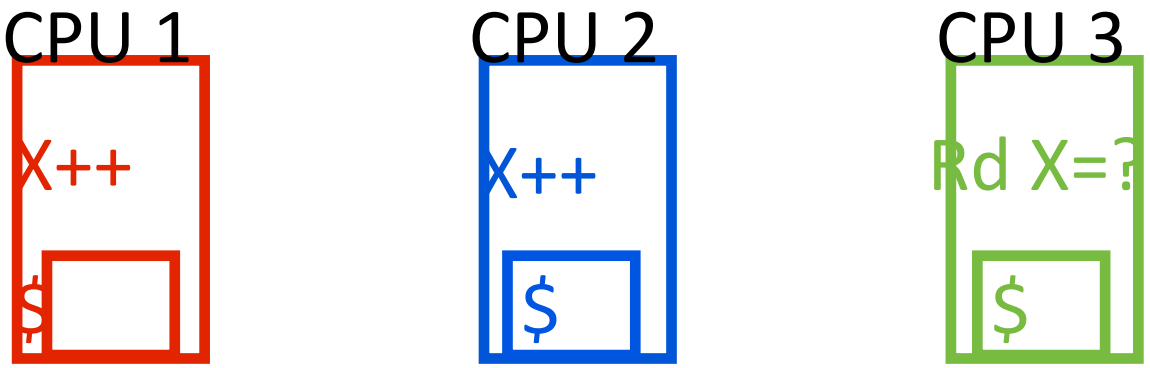
X++

Rd X=**2**

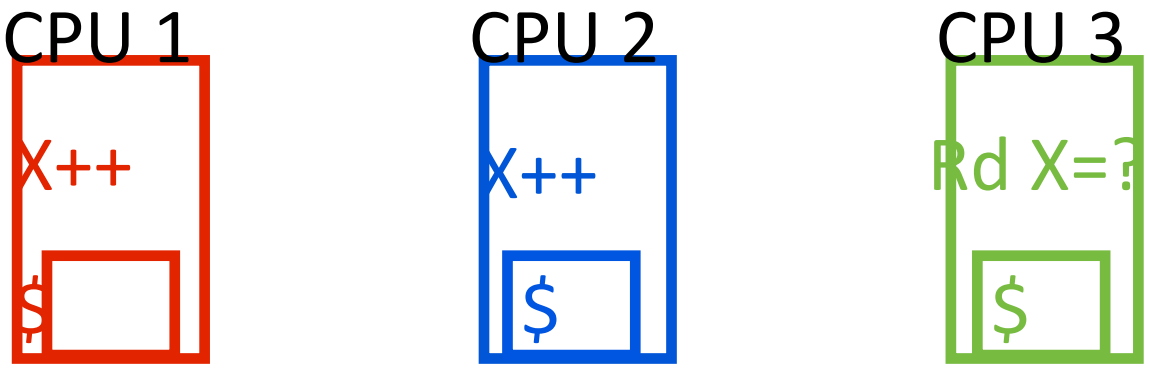
Rd X=**2**

Rd X=**1**

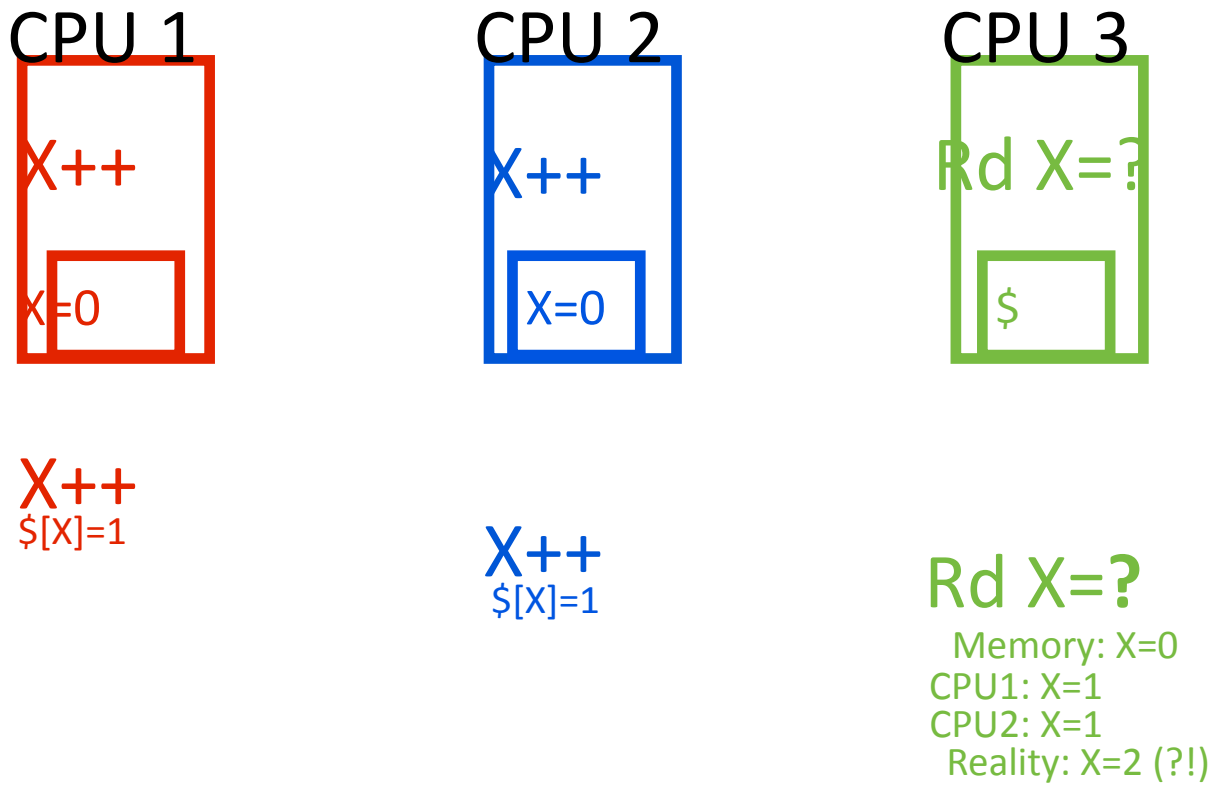
(and the symmetric case)



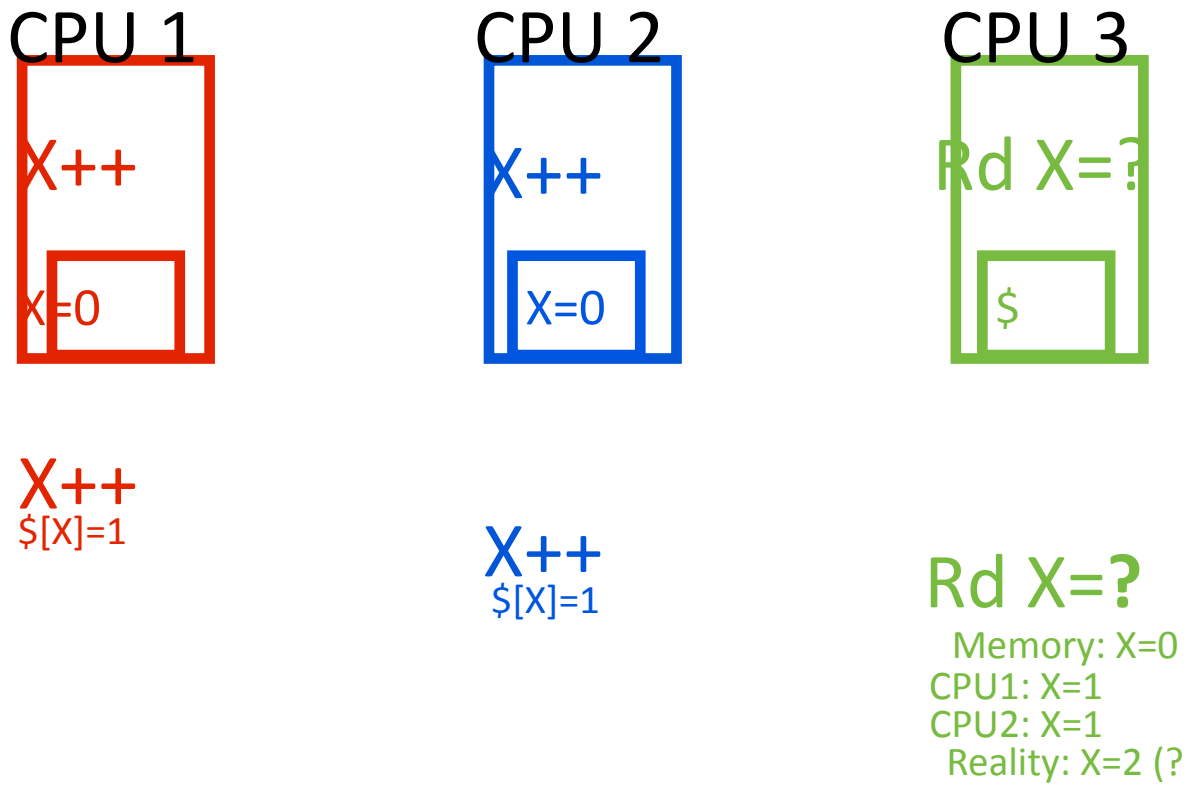
What assumptions are we making about the system to produce the results 0, 1, and 2?



We assume the updates see one anothers' results!
(Why wouldn't they?)



So what the heck do we do 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"

Informally Defining Coherence

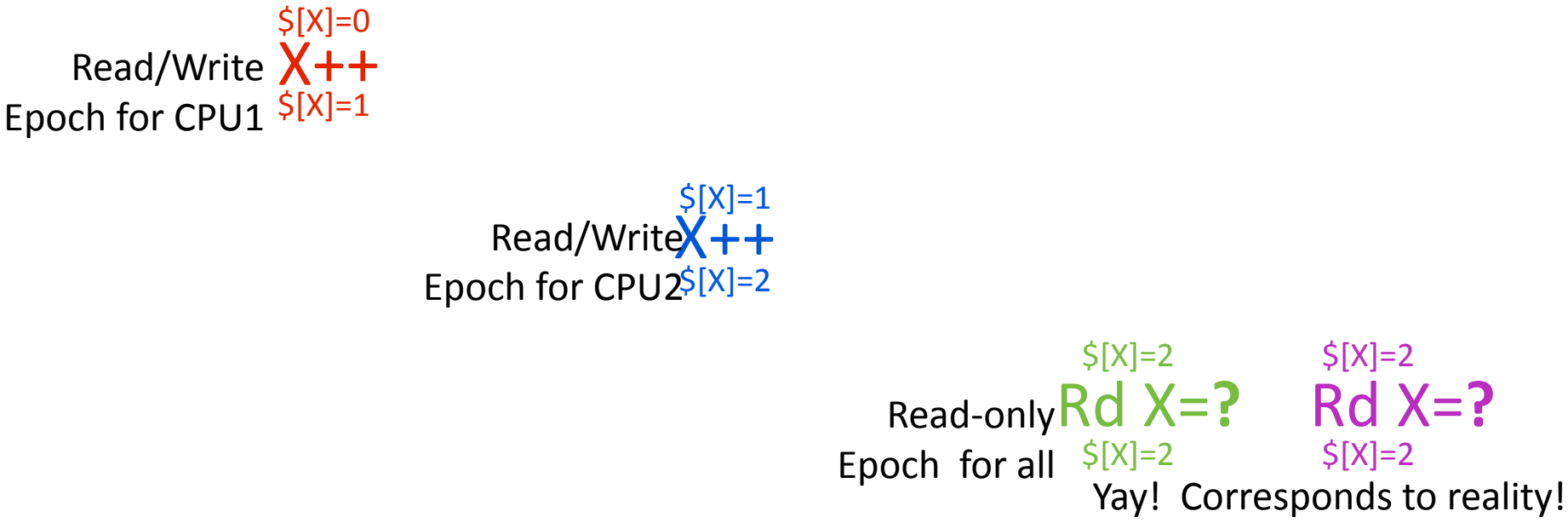
> “Coherence serializes all reads with all updates to the same location by different CPUs/caches, so that each read sees the result of the most recent update by any other”

“Single Writer/Multiple Reader (SWMR) Invariant

+

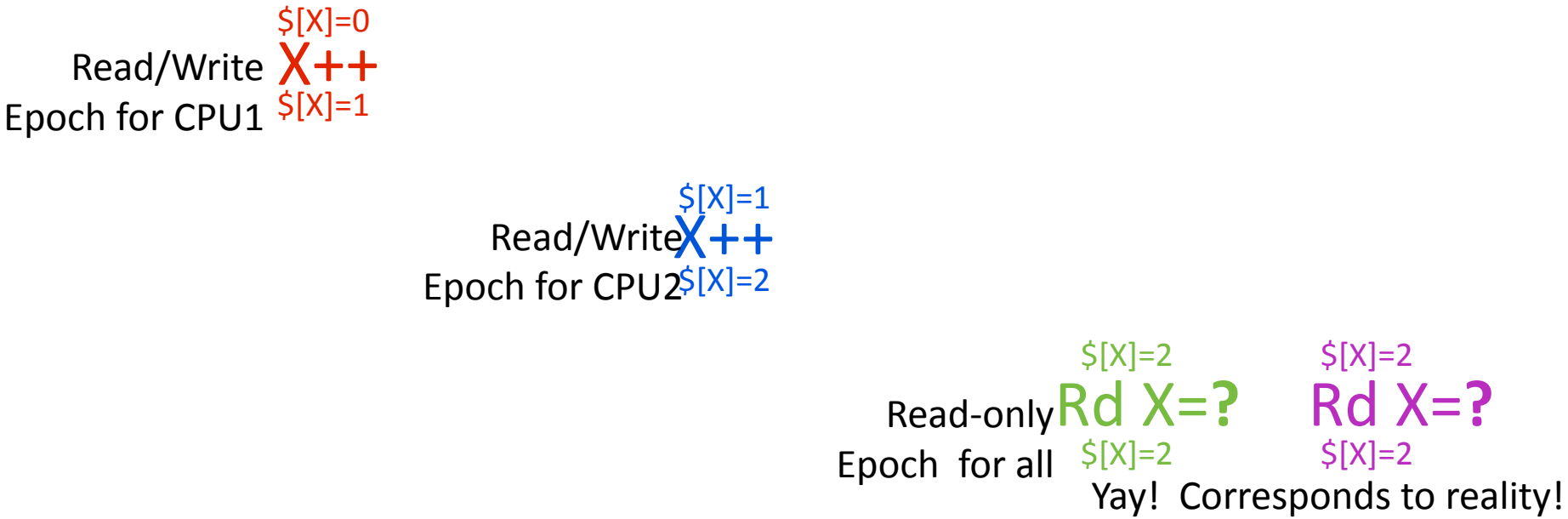
Data-Value Invariant”

Epoch Model


$[X]=0
Read/Write X++
$[X]=1
Epoch for CPU1
$[X]=1
Read/Write X++
$[X]=2
Epoch for CPU2
$[X]=2
$[X]=2
Read-only Rd X=? Rd X=?
Epoch for all
$[X]=2
$[X]=2
Yay! Corresponds to reality!

Epoch Model

R/W vs. R-O Epochs directly enforce SWMR



Epoch transitions assume data-value invariant

What do we need to implement the Epoch Model?

Need to add concept of R/W epoch vs. R-O epoch

Need to add gadget that correctly moves data between epochs

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

(1) CPU1 says “I am is writing X”
Others relinquish cached copies of X
and reply “OK go for it” <enter R/W epoch>

(ditto (1) for CPU2) (2)

$[X]=0
Read/Write X++
$[X]=1
Epoch for CPU1

$[X]=1
Read/Write X++
$[X]=2
Epoch for CPU2

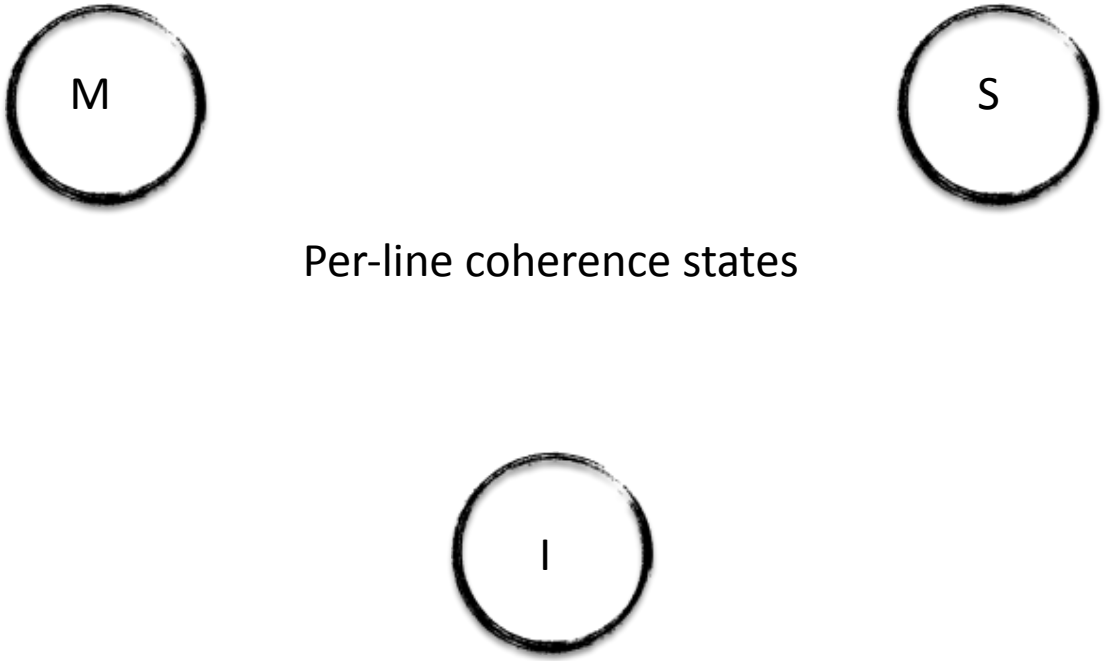
CPU3 says “I want to read only”
(4) Others reply “OK, we all agree not to write without saying so”
<enter R-O epoch>

$[X]=2 $[X]=2
Read-only Rd X=? Rd X=?
Epoch for all $[X]=2 $[X]=2

CPU1 replies “I have X. Use my copy or get it from memory after I write it back” (3)

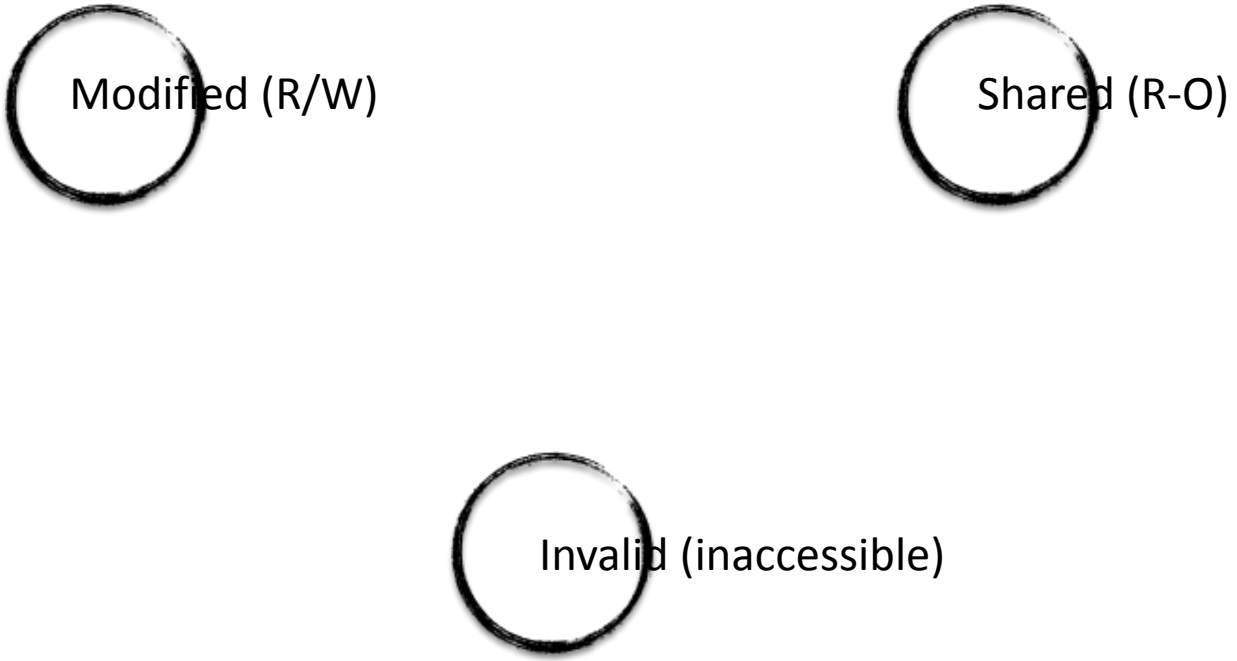
(ditto (3) for CPU 2) (5)

Cache Coherence Protocol



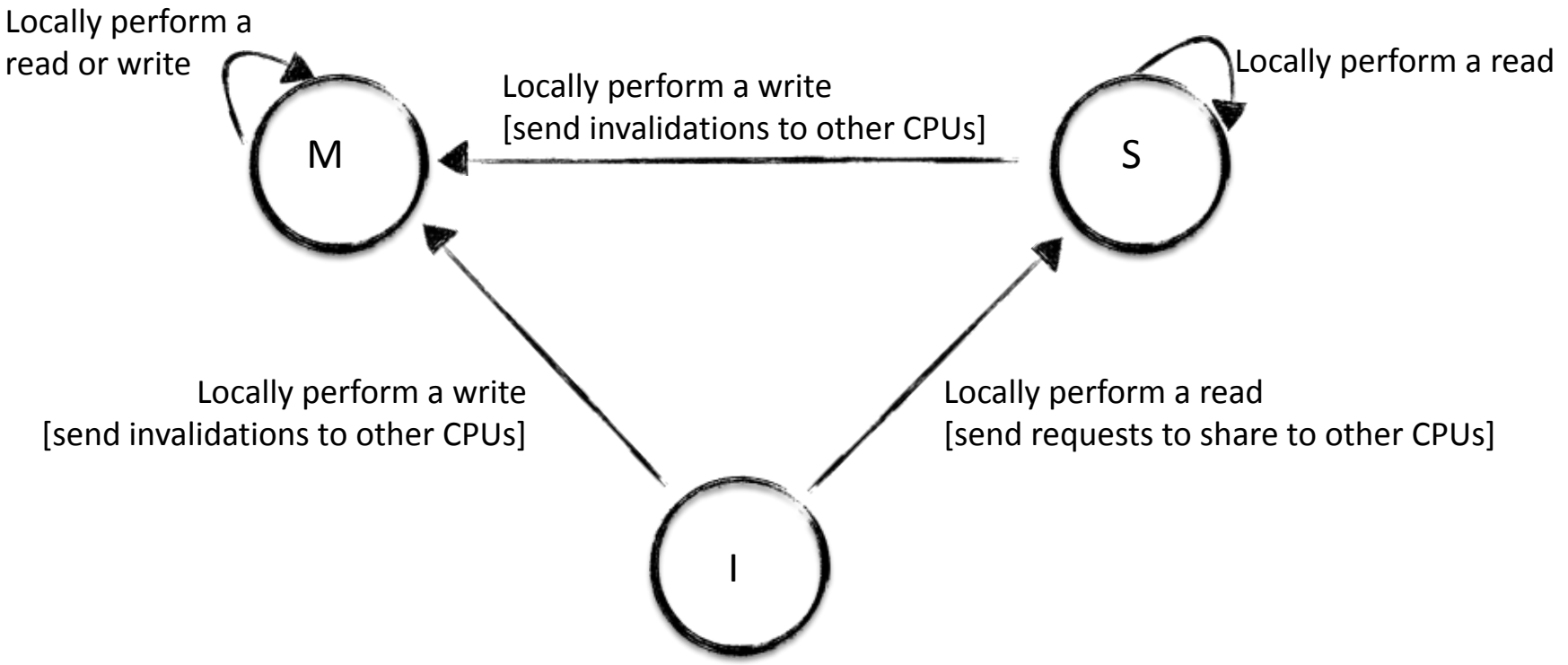
Per-line coherence states

Cache Coherence Protocol



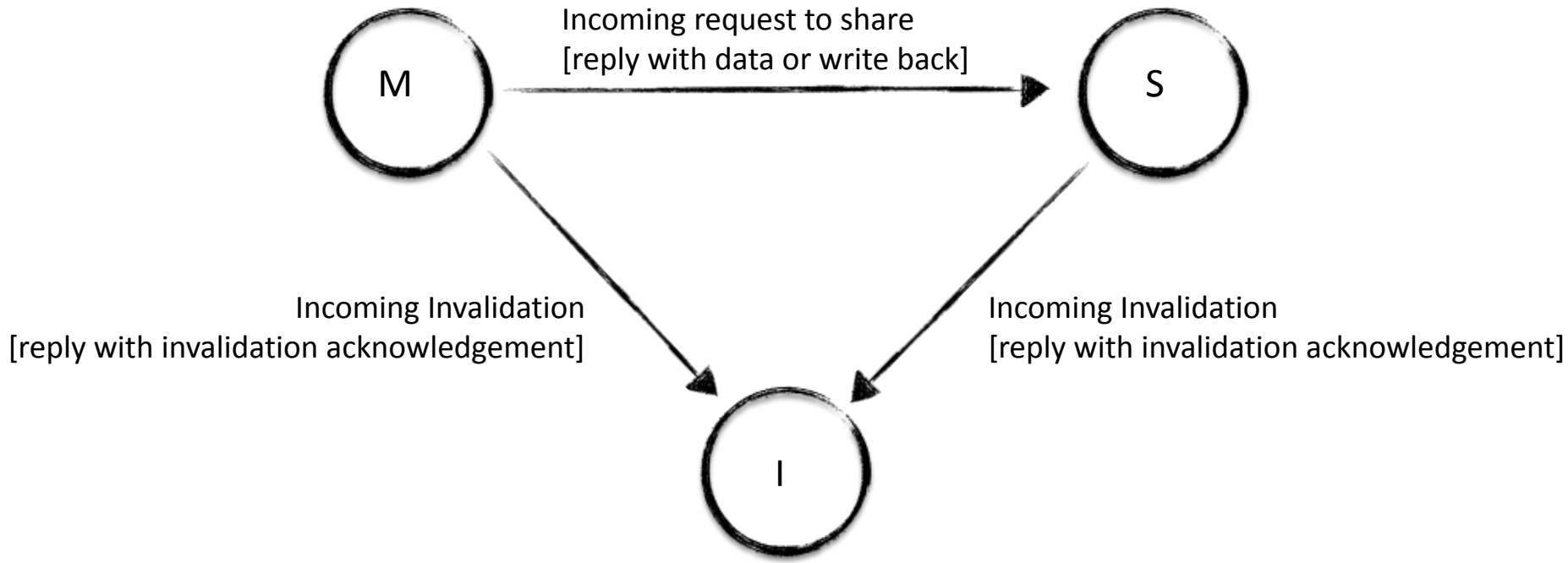
Cache Coherence Protocol

Local operations perspective

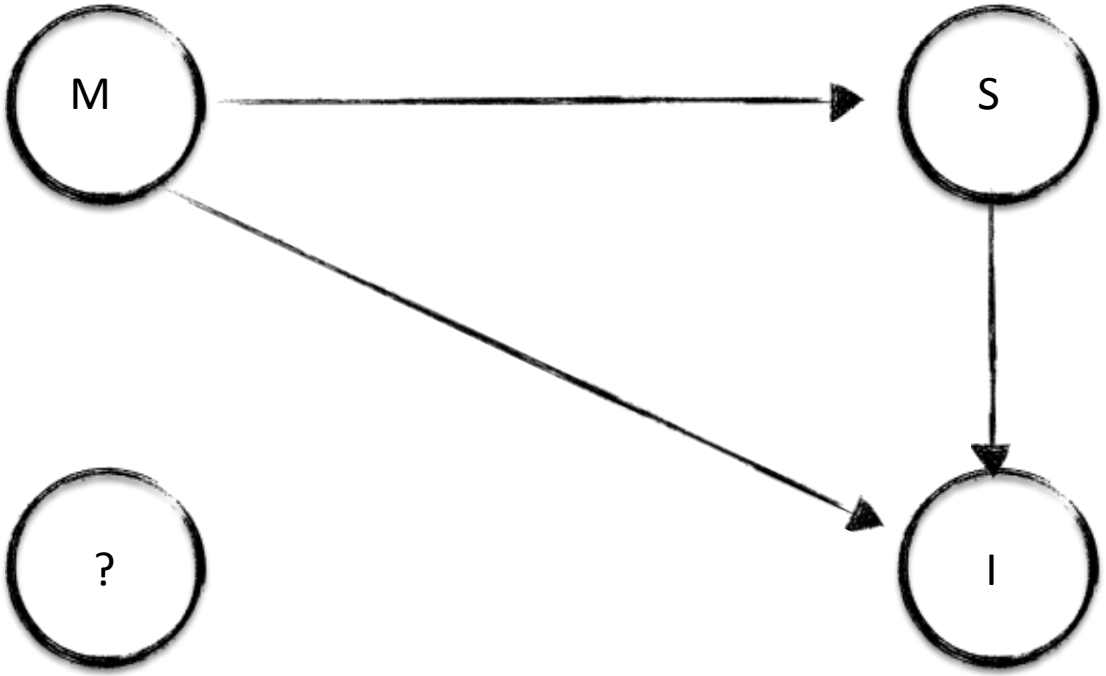


Cache Coherence Protocol

Remote operations perspective

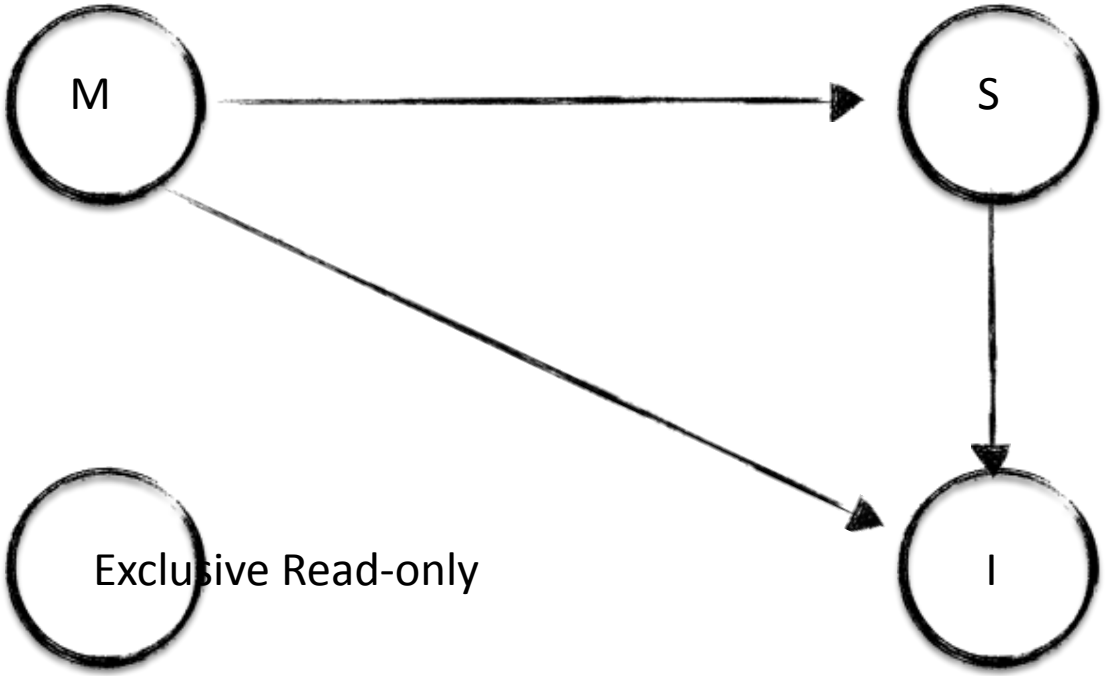


Can we design another state?



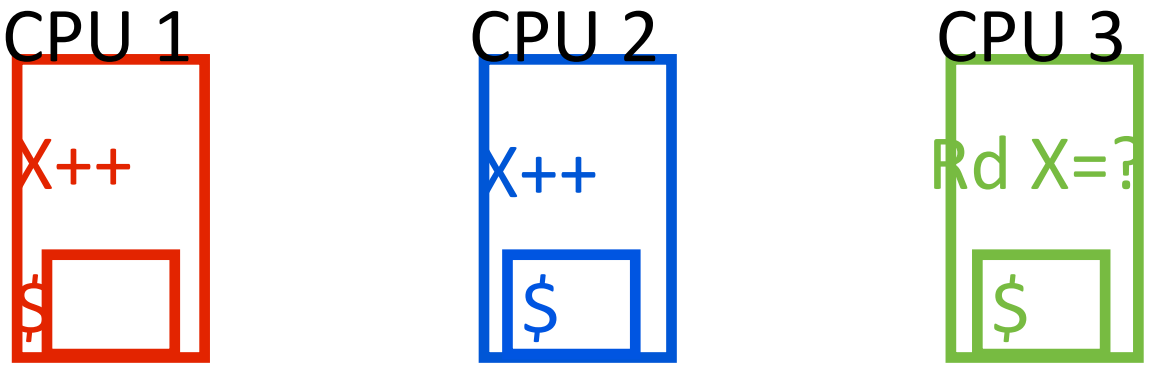
What should we optimize?

Can we design another state?

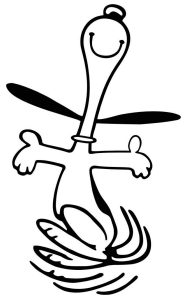


(Benefit: no invalidation required
to transition from E->M, like from S->M)

Implementing the Protocol

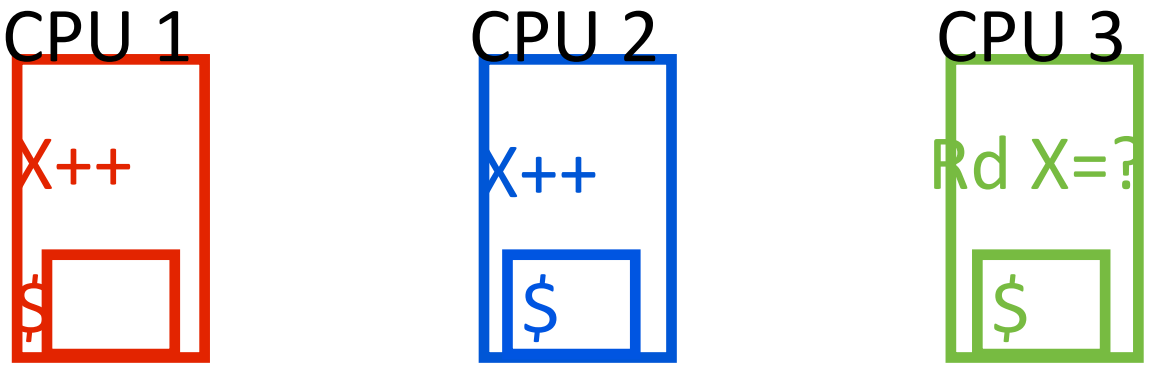


Shared bus for coherence messages



Snoopy Coherence

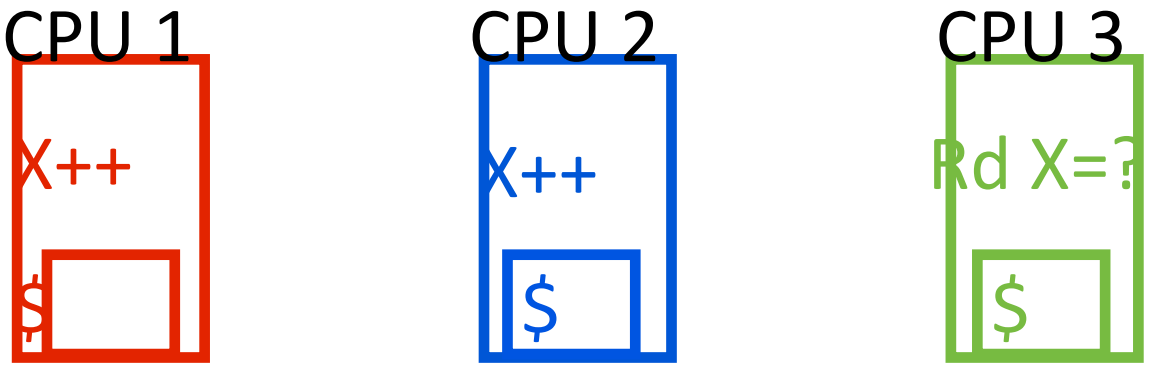
Implementing the Protocol



Invalidate

X++

Implementing the Protocol

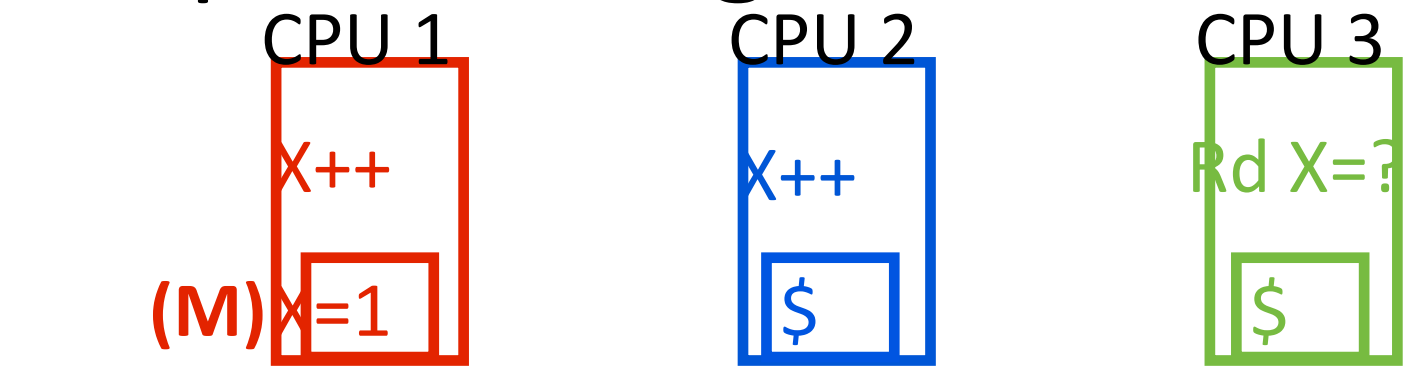


Ack

Ack

X++

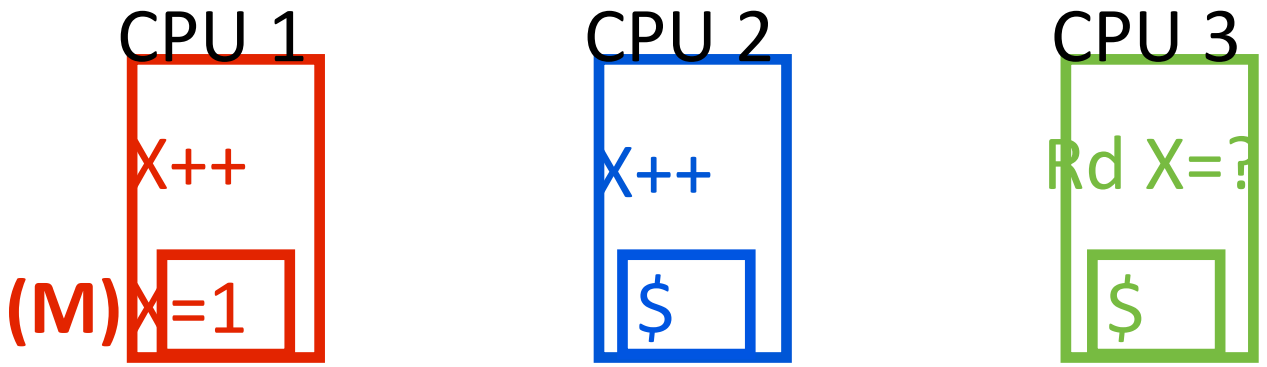
Implementing the Protocol



Entering CPU1's write epoch

X++

Implementing the Protocol

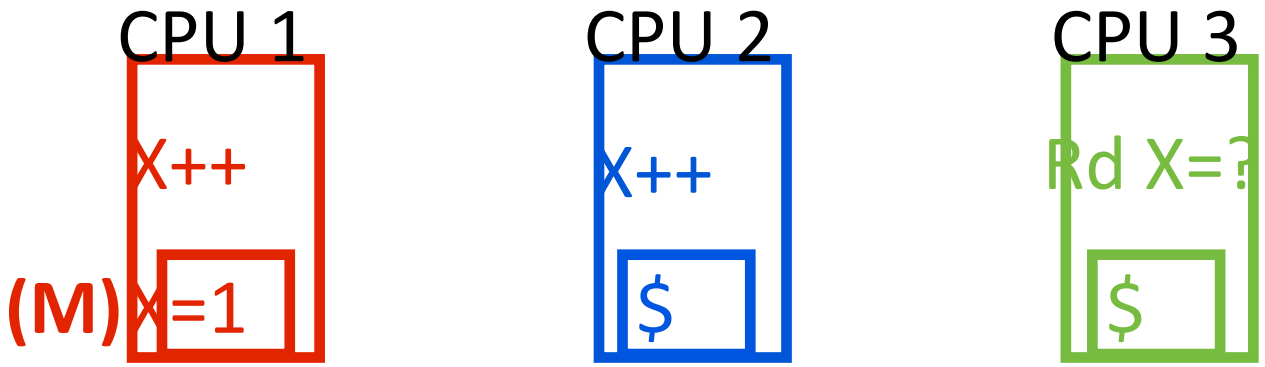


RdReq

X++

Rd X=?

Implementing the Protocol



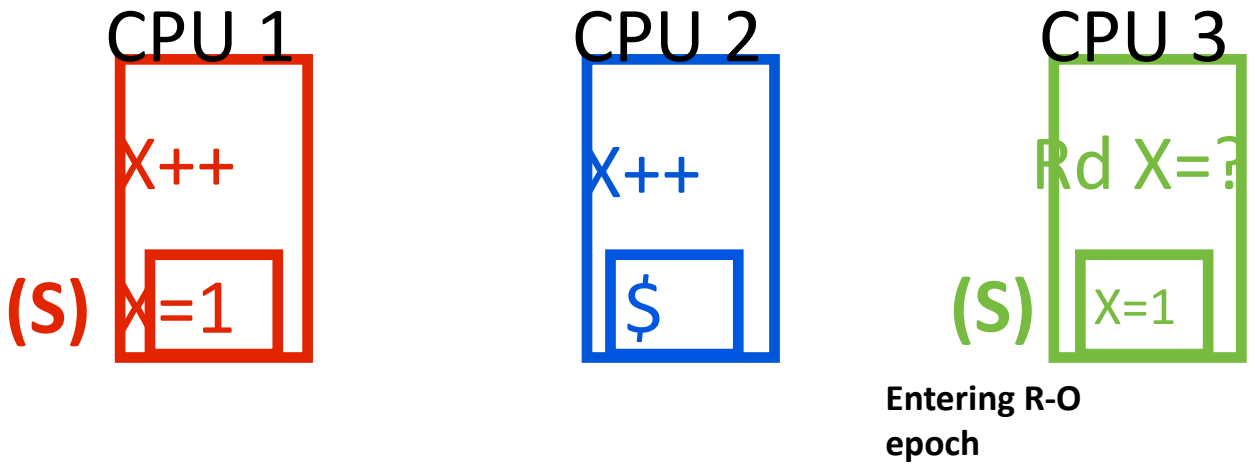
Got it: X=1

Don’t have it

X++

Rd X=?

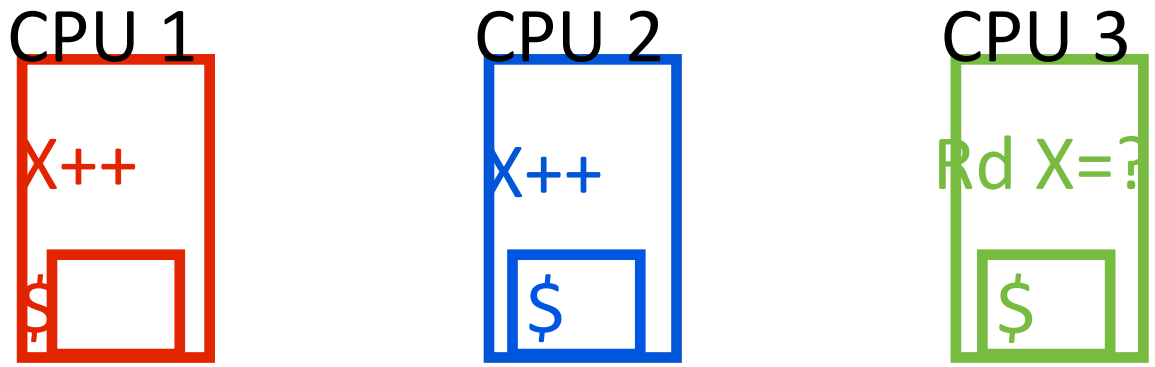
Implementing the Protocol



X++

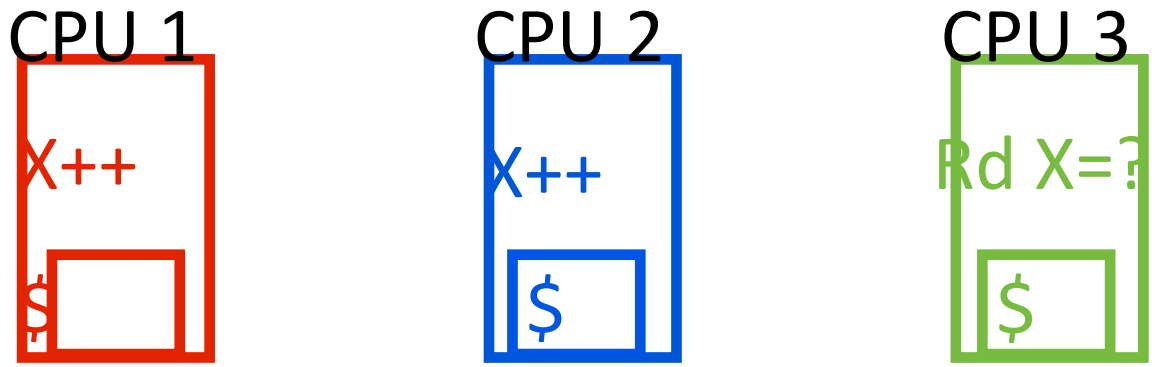
Rd X=?

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

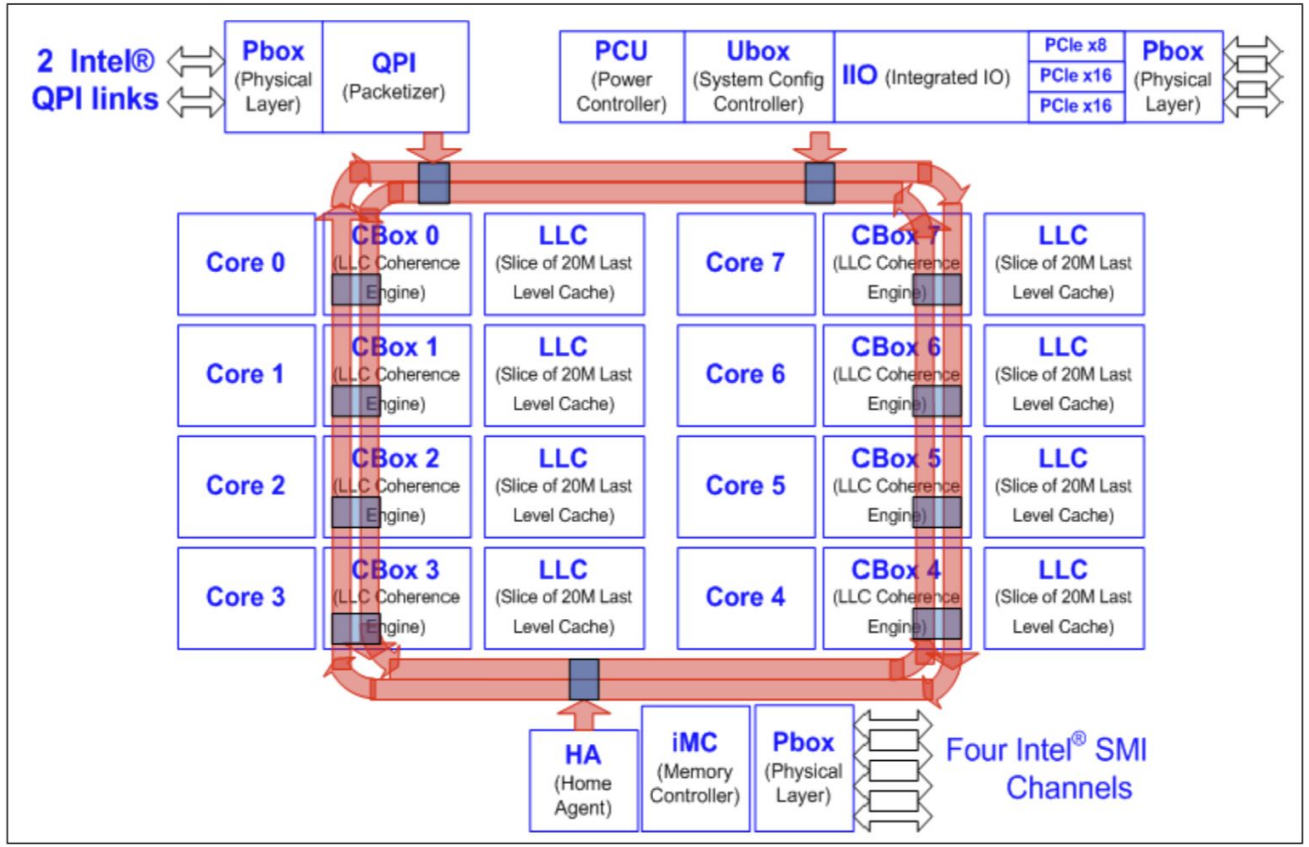
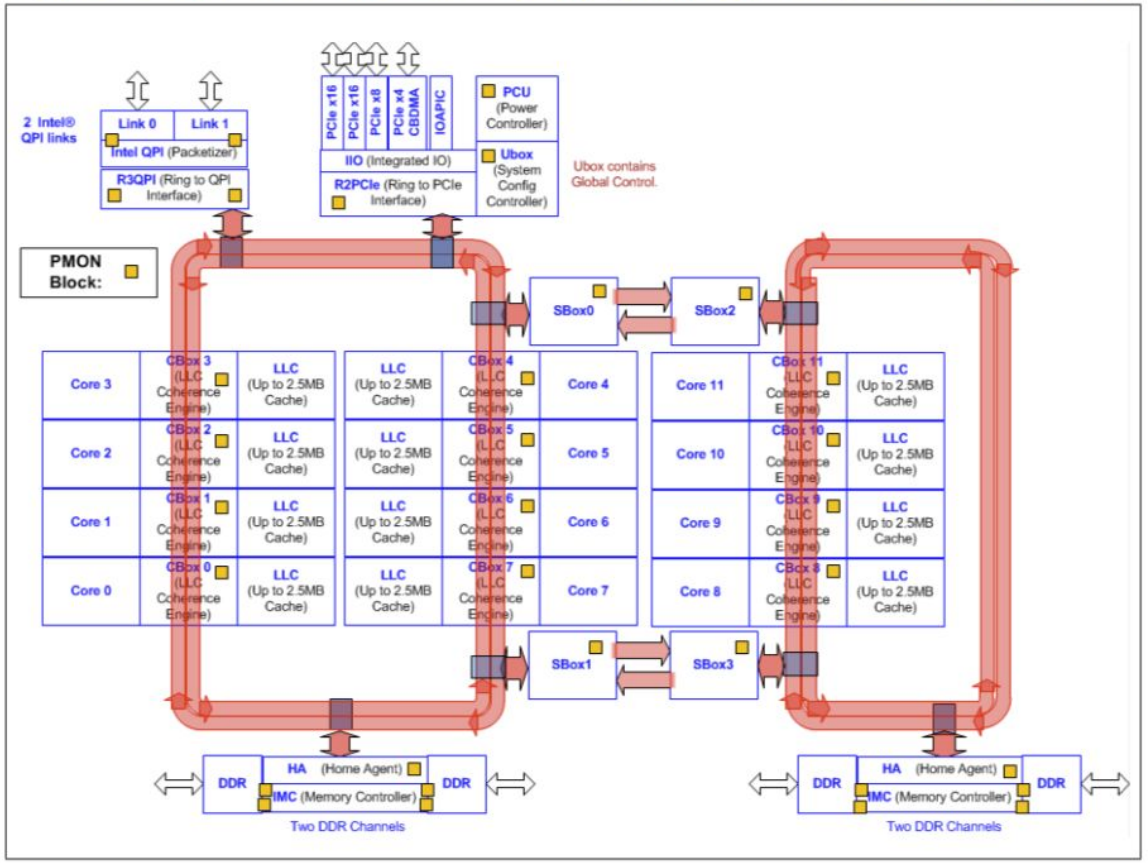
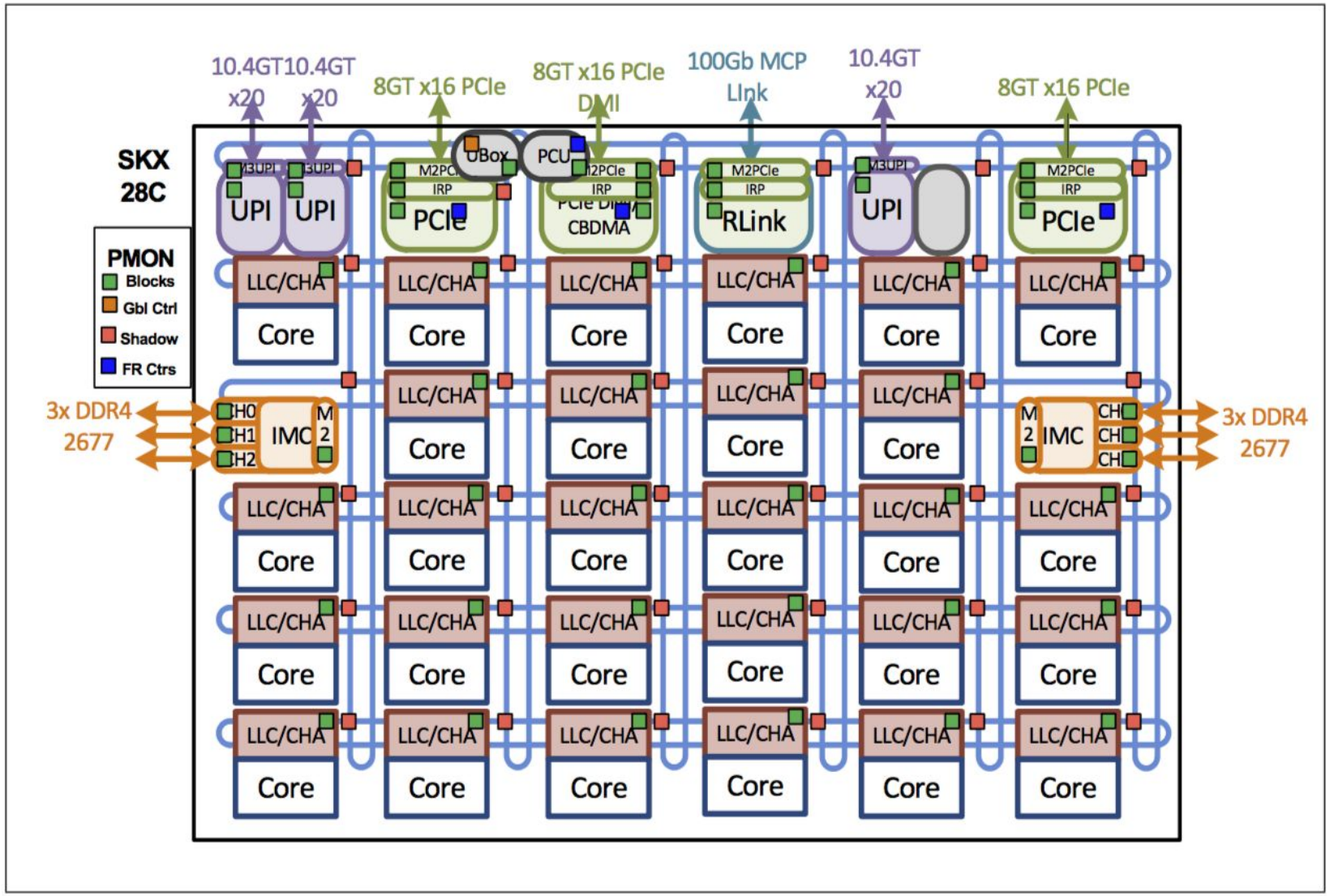


Figure 1-2. Intel® Xeon® Processor E5 v3-1600/2600/4600 Family -12C Block Diagram



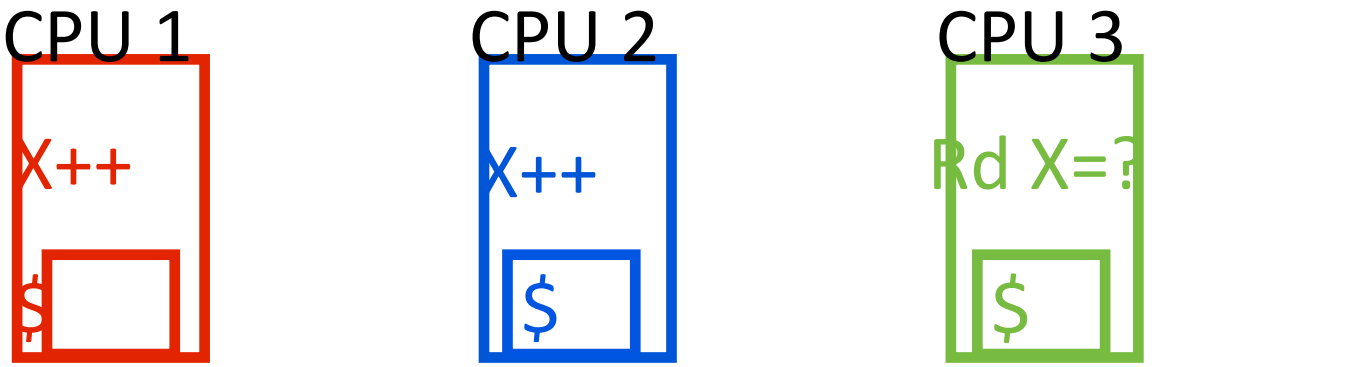
Intel Sandybridge Multiprocessor: bi-directional ring network

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



Skylake Xeon 2017 2D mesh

Implementing the Protocol

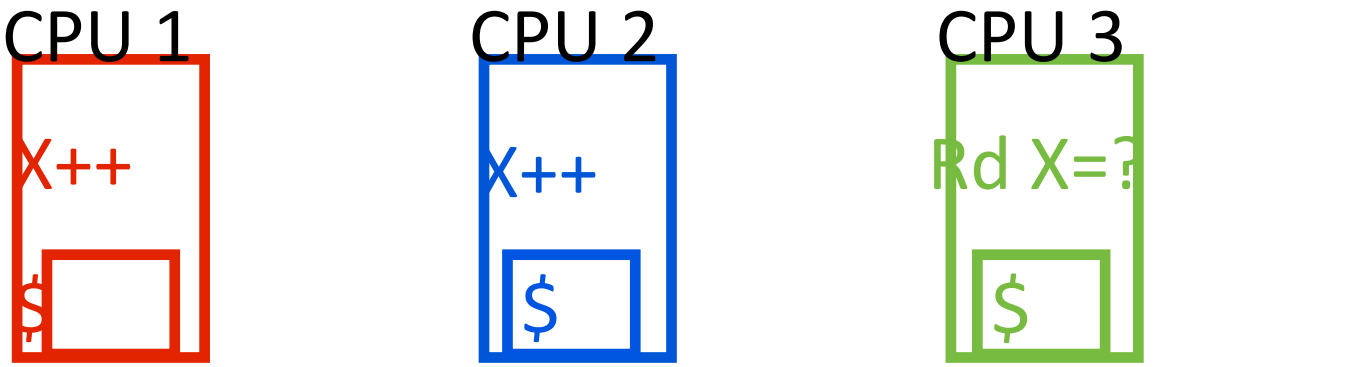


(Effectively) Point to Point Links

Sharers of X

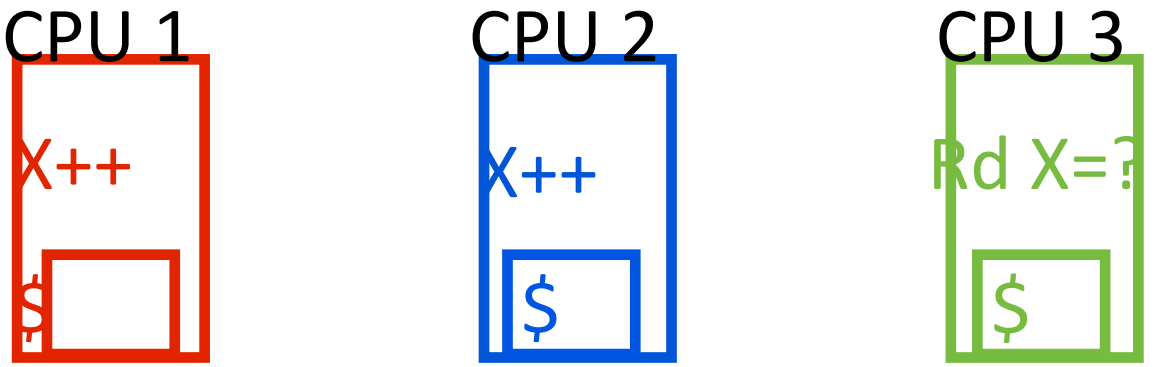
Directory-based

Implementing the Protocol



Directory-based

Implementing the Protocol



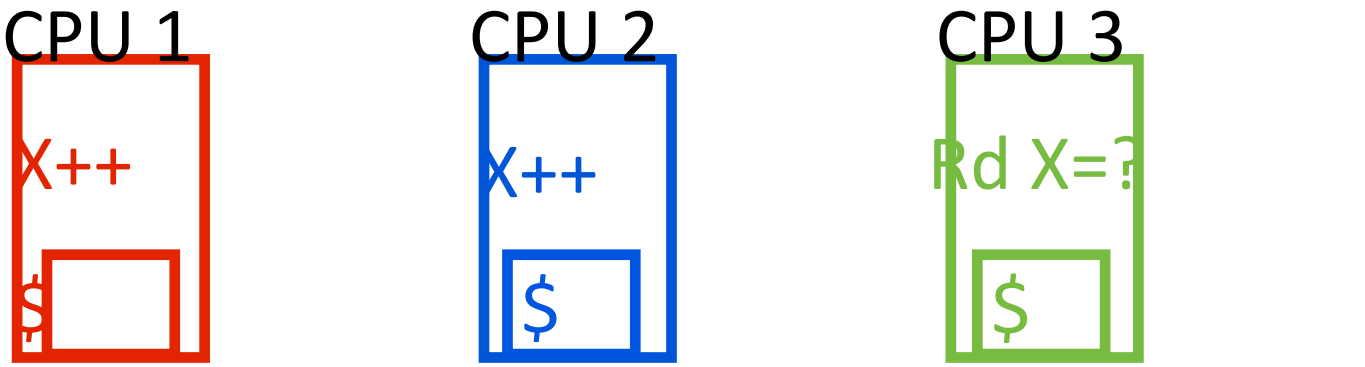
Who has X?

Sharers of X

X++

Directory-based

Implementing the Protocol



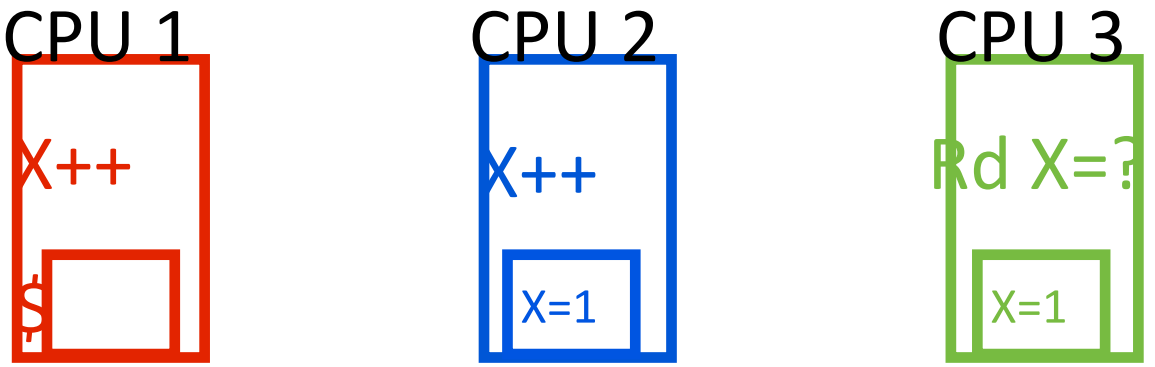
No one does!
Proceed!

Sharers of X

X++

Directory-based

Implementing the Protocol



CPUs 2 and 3 do.
Send them Invalidates!

Sharers of X

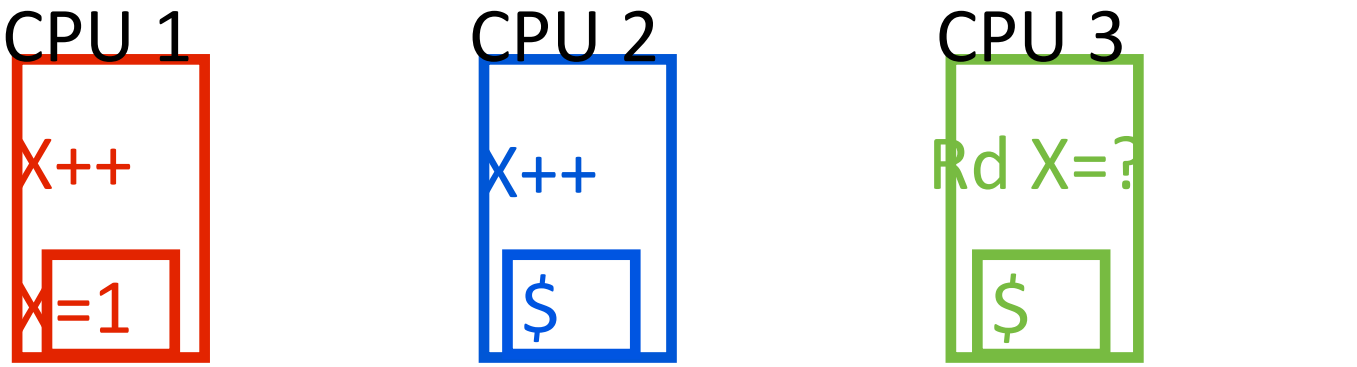
X++

Rd X=?

X++

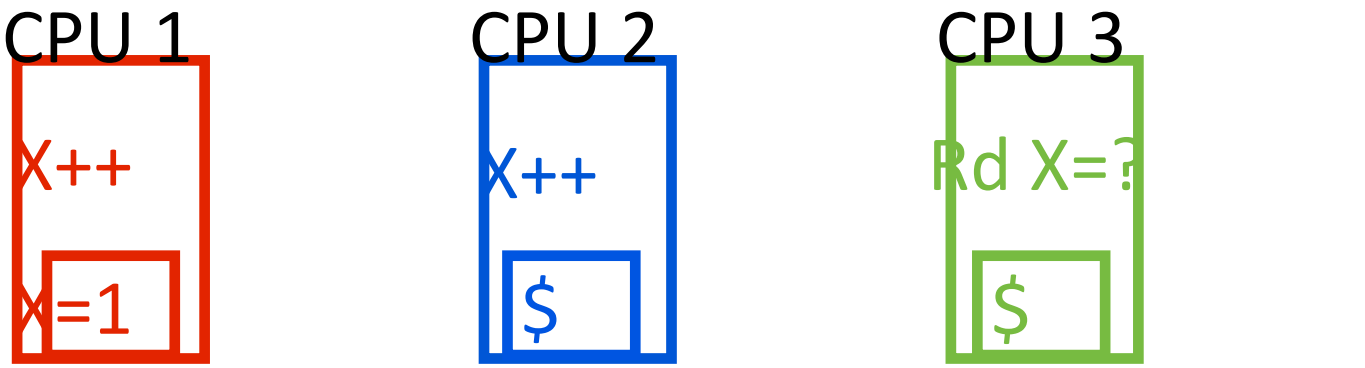
Directory-based

Implementing the Protocol



Benefit: No broadcast on shared bus

Implementing the Protocol

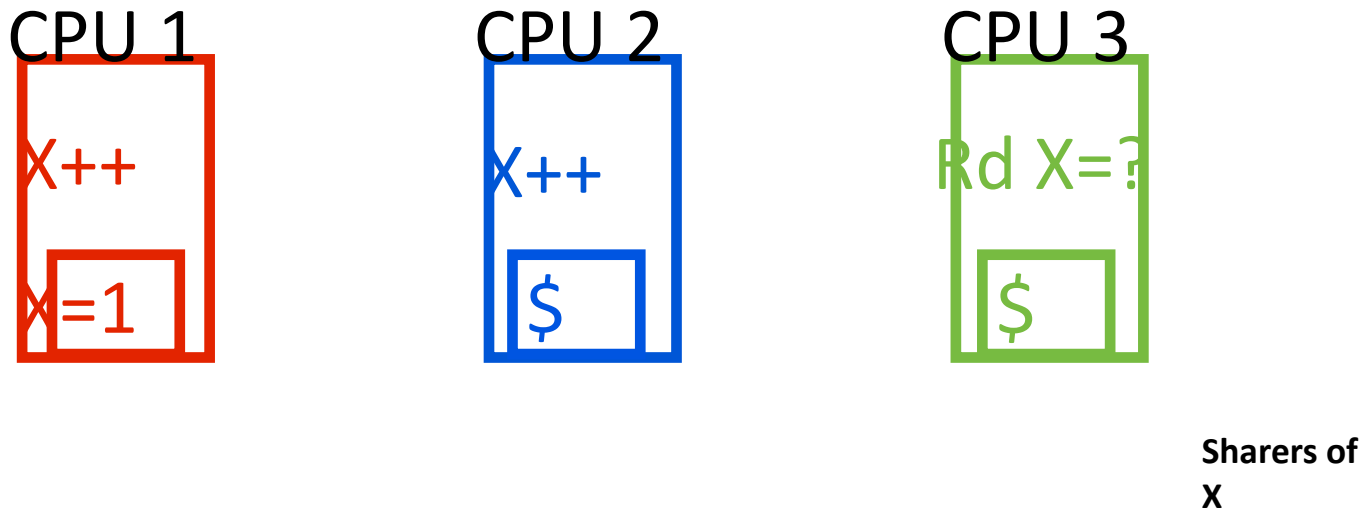


Sharers of X

X++

Drawbacks?

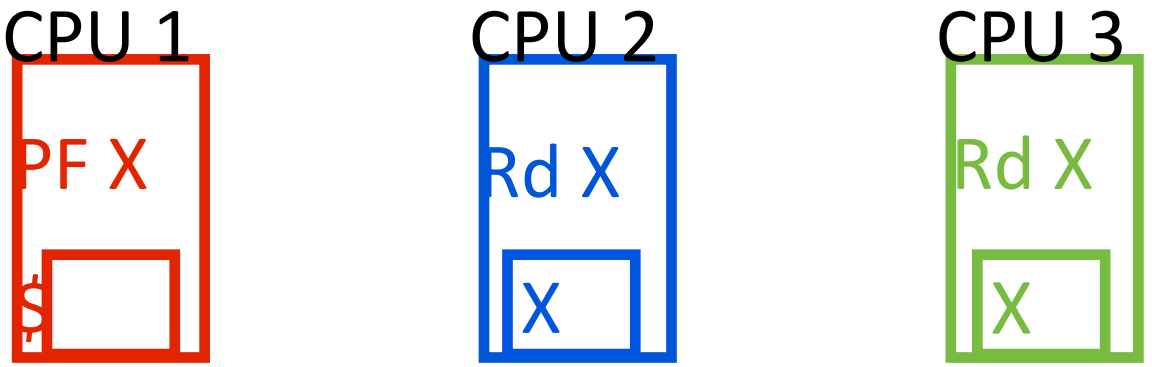
Implementing the Protocol



Centralized directory won’t scale

(In Practice: Distribute Directory)

Optimization: Non-binding Prefetch



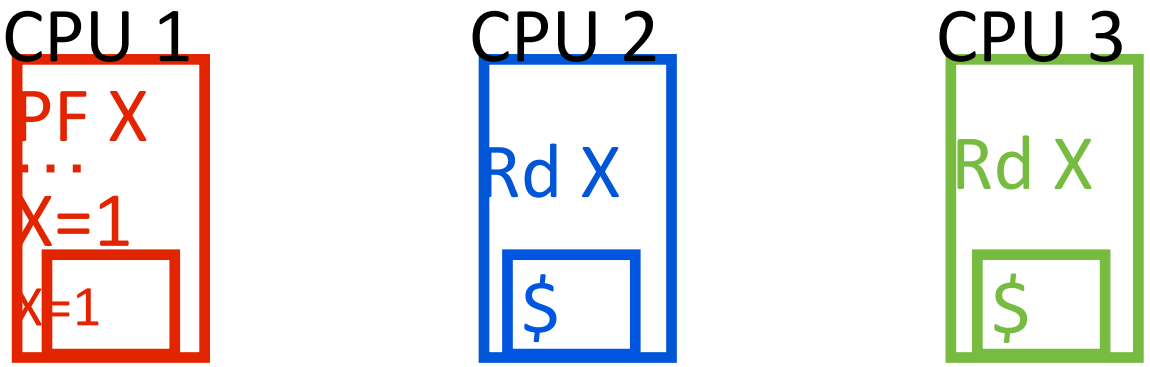
Sharers of X

CPU 2

CPU 3

Prefetch instruction preemptively changes coherence state

Optimization: Non-binding Prefetch



Owner of X

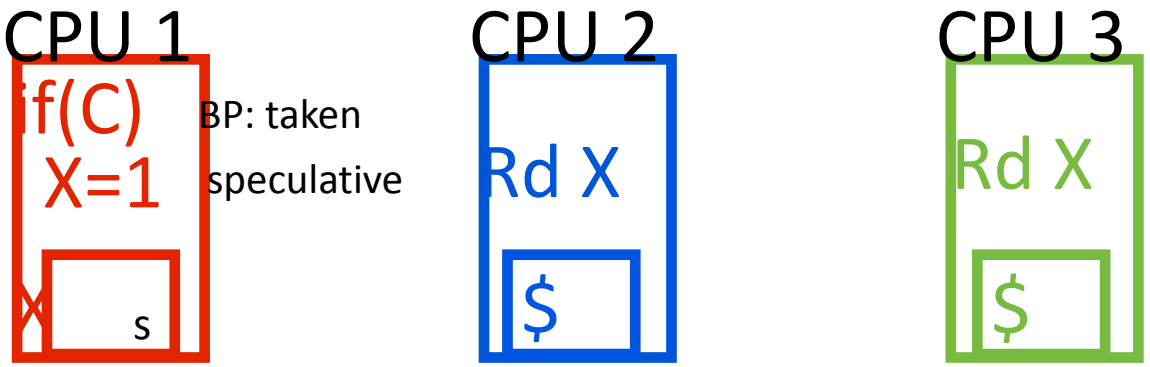
~~CPU 2~~

~~CPU 3~~

CPU 1

Benefit?

Optimization: Speculation



Owner of X

~~CPU 2~~

~~CPU 3~~

CPU 1

Speculative operations that squash behave like non-binding pre-fetch

“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 Definition:

> “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

Wr X

Wr X

OR

Wr X

Wr X

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

Consistency is Ordering

Wr X

Wr Y

OR

Wr Y

Wr X

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

Sequential Consistency (SC)

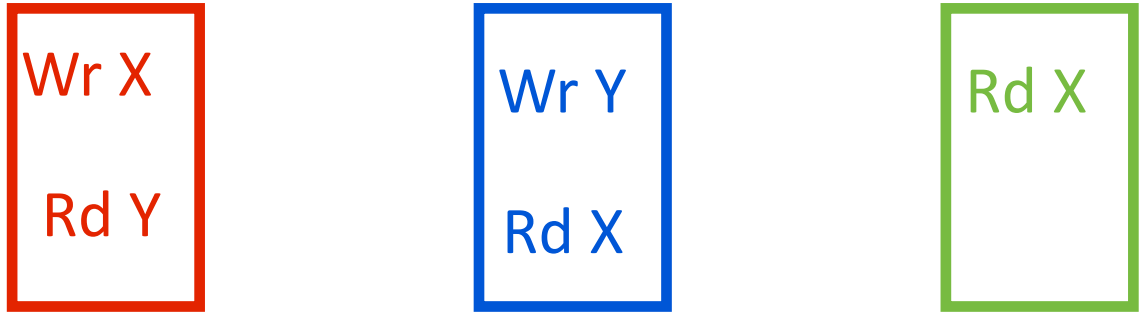
The simplest, most intuitive memory consistency model

Two Invariants to SC:

Invariant #1:
Instructions are executed in program order

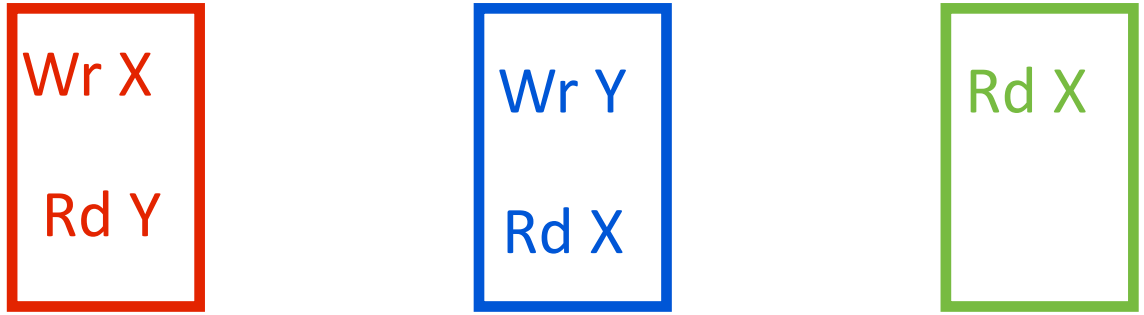
Invariant #2:
All processors agree on a total order of executed instructions

The SC “Switch”



Execution

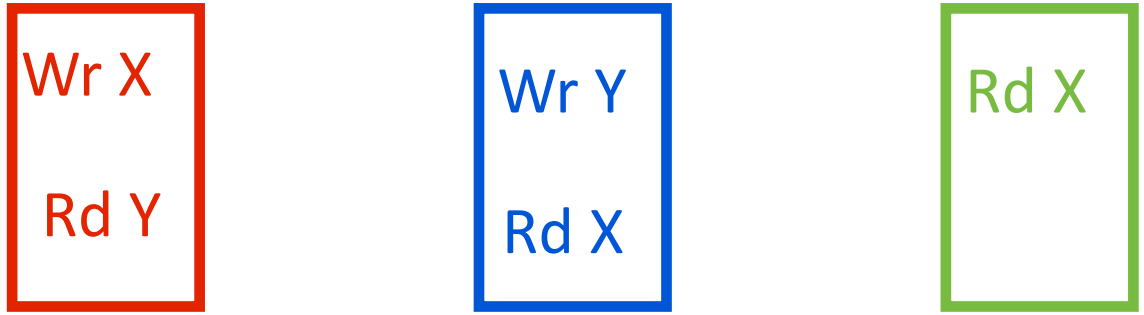
The SC “Switch”



Execution

Wr X

The SC “Switch”

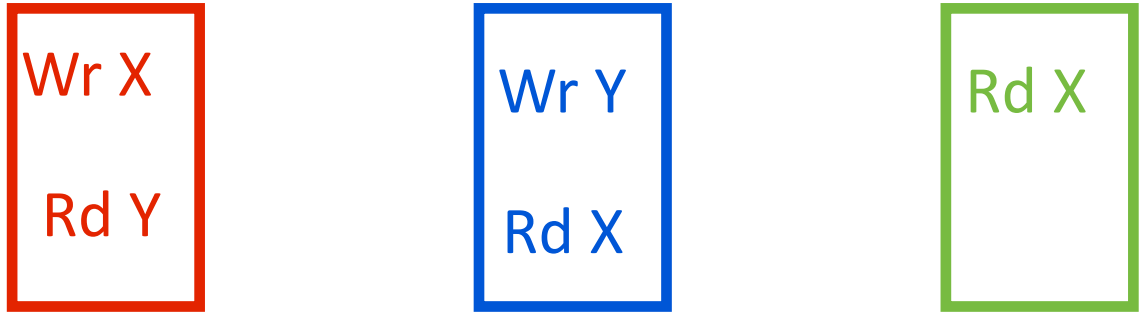


Execution

Wr X

Rd Y

The SC “Switch”



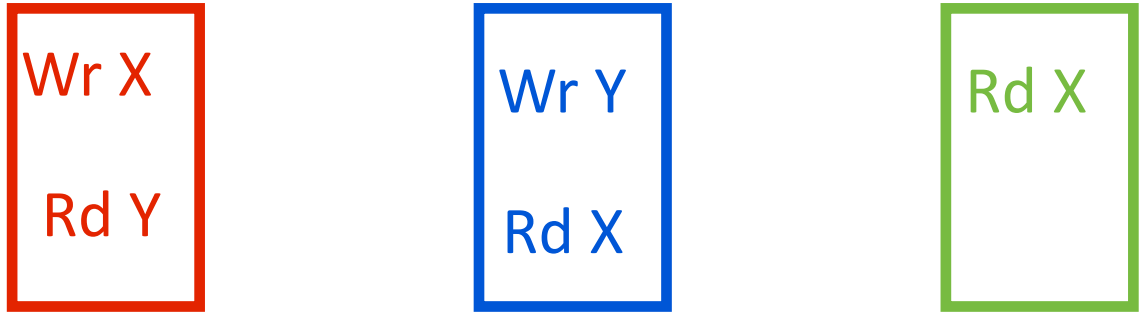
Execution

Wr X

Rd Y

Wr Y

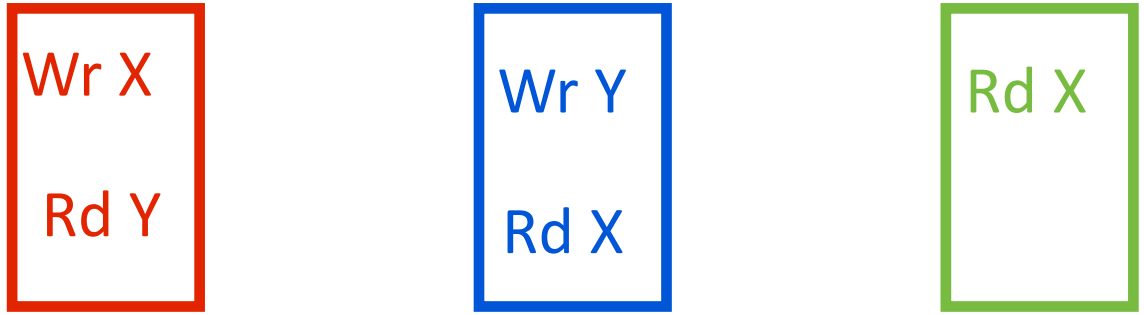
The SC “Switch”



Execution

Wr X
Rd Y
Wr Y
Rd X

The SC “Switch”

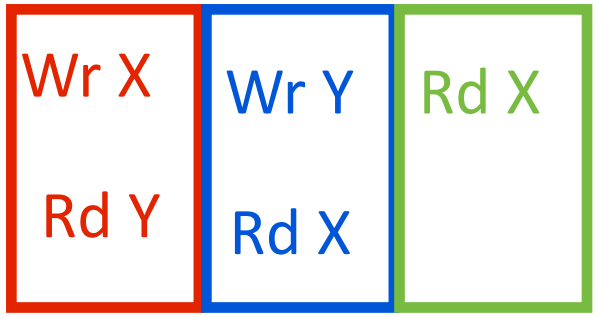


Execution

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.



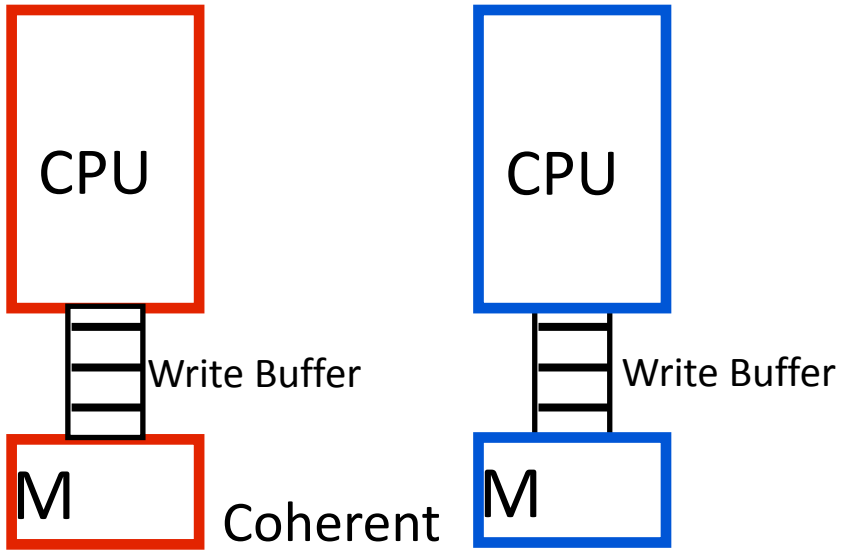
Intuitive (SC)	Weird (not SC)
Wr X	Rd Y
Rd Y	Wr X
Wr Y	Rd X
Rd X	Rd X
Rd X	Wr Y

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

Why are Instructions Reordered?

Examples? When does it matter? When does it not matter?

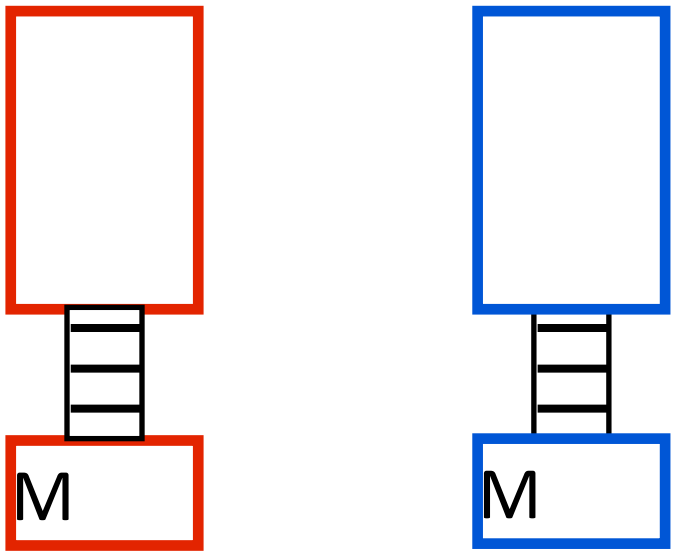
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



Program

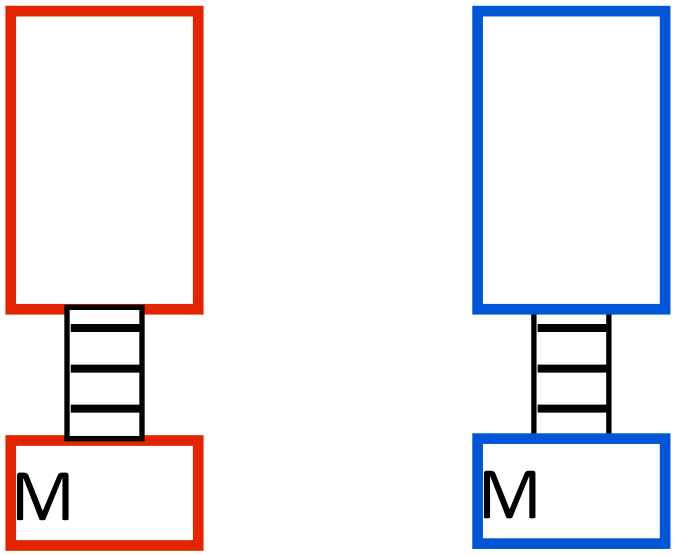
Initially X == Y == 0

X=1 Y=1

r1=Y r2=X

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

Reordering #1: Write Buffers



Program

Initially X == Y == 0

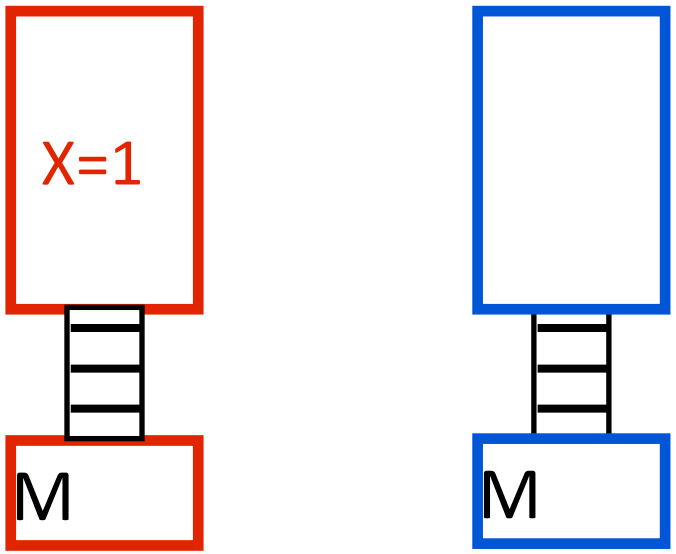
X=1 Y=1

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

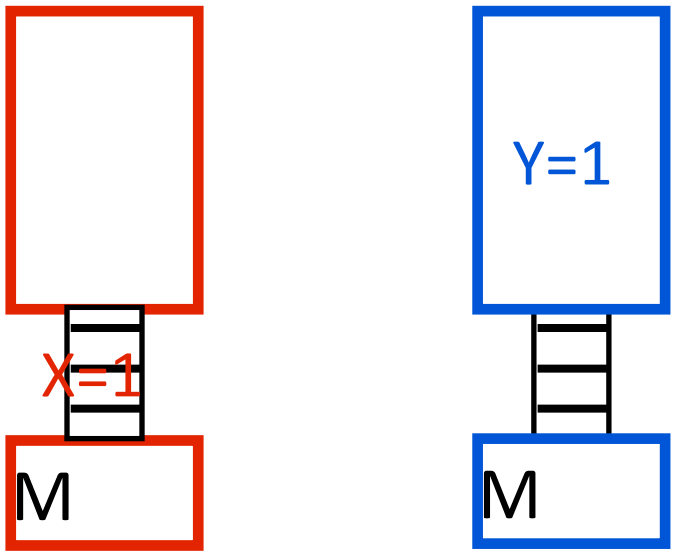
Initially X == Y == 0

Y=1

r1=Y r2=X

Execution

Reordering #1: Write Buffers



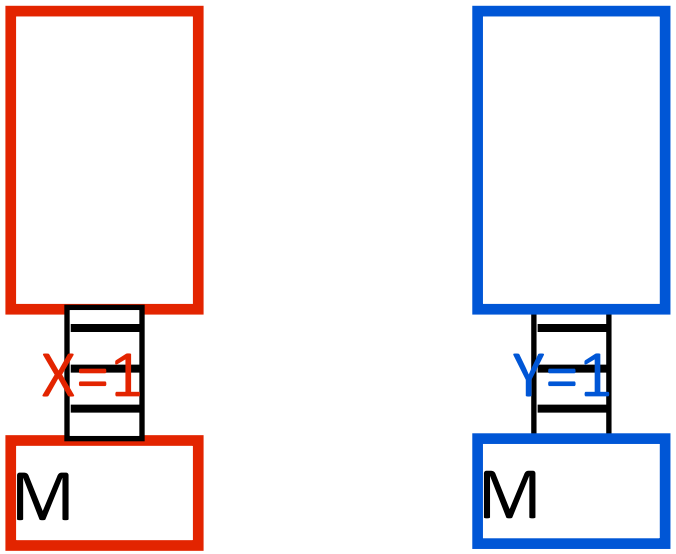
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



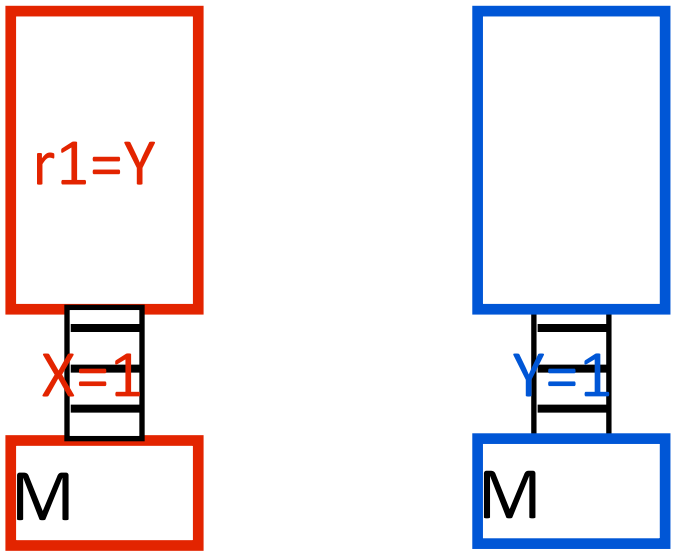
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



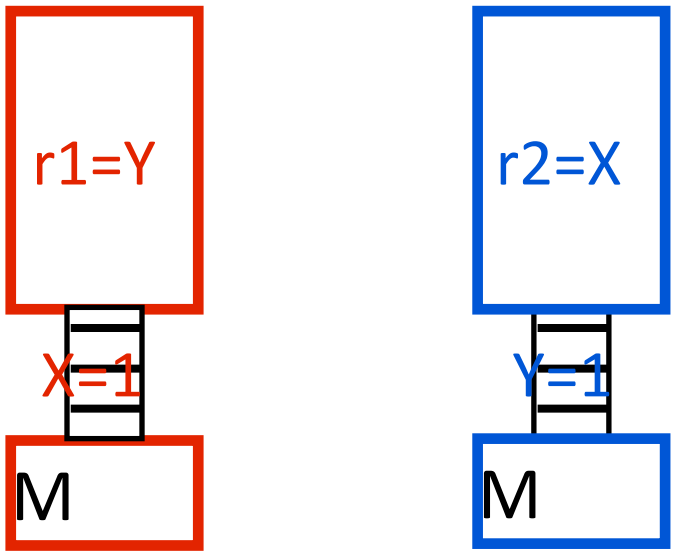
Program

Initially X == Y == 0

r2=X

Execution

Reordering #1: Write Buffers

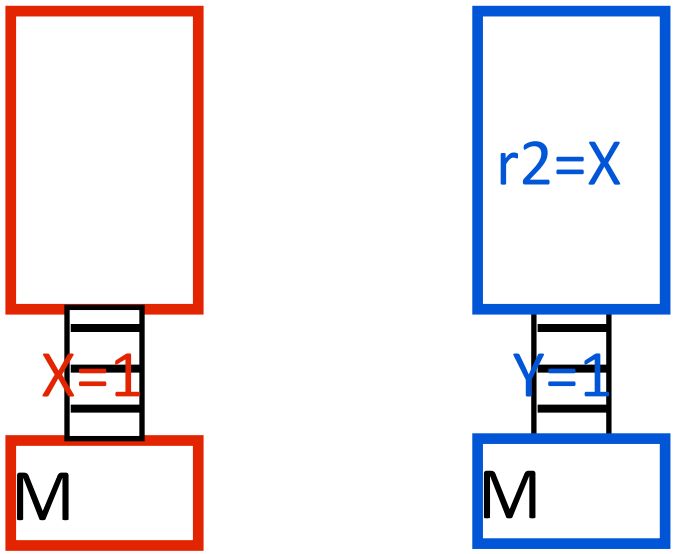


Program

Initially X == Y == 0

Execution

Reordering #1: Write Buffers



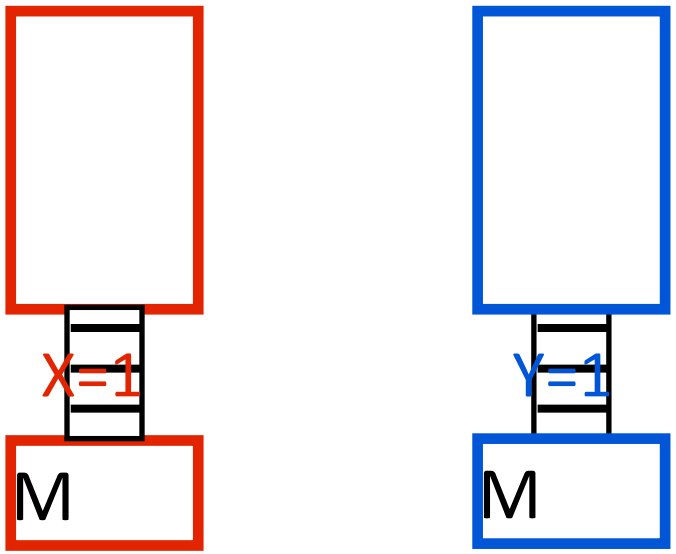
Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

Reordering #1: Write Buffers



Program

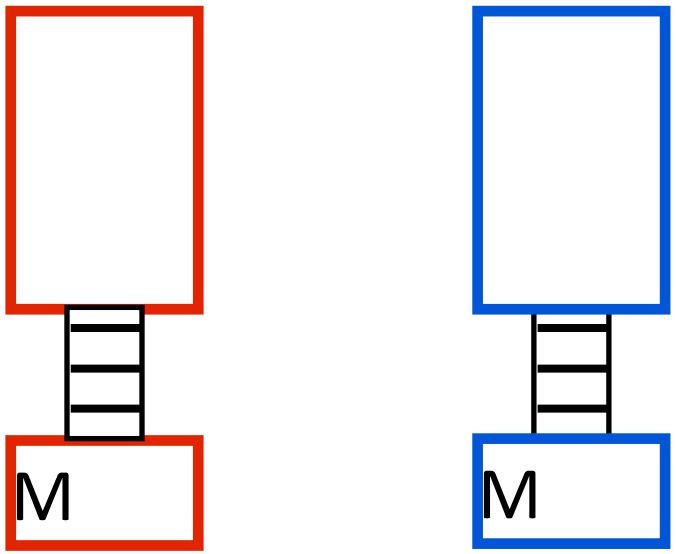
Initially X == Y == 0

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]

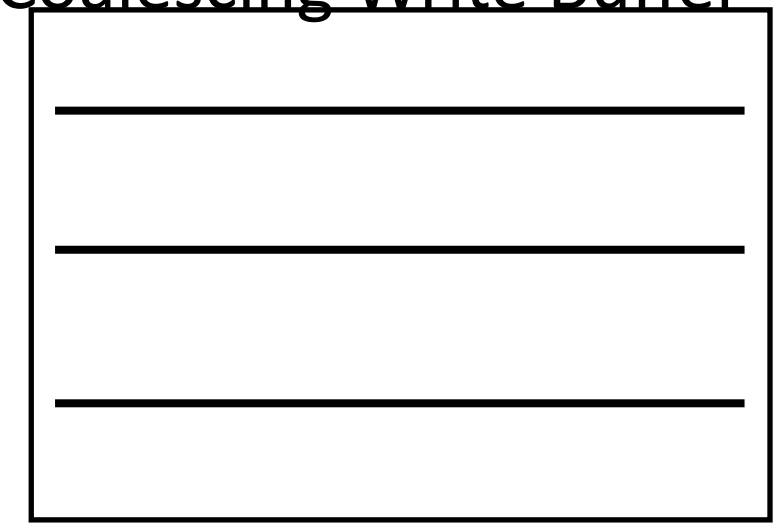
r2=X [r2 <- 0]

X=1

Y=1 (Not SC!)

Reordering #2: Write Combining

Coalescing Write Buffer



4 word cache line

Program

X,Z in same $ line

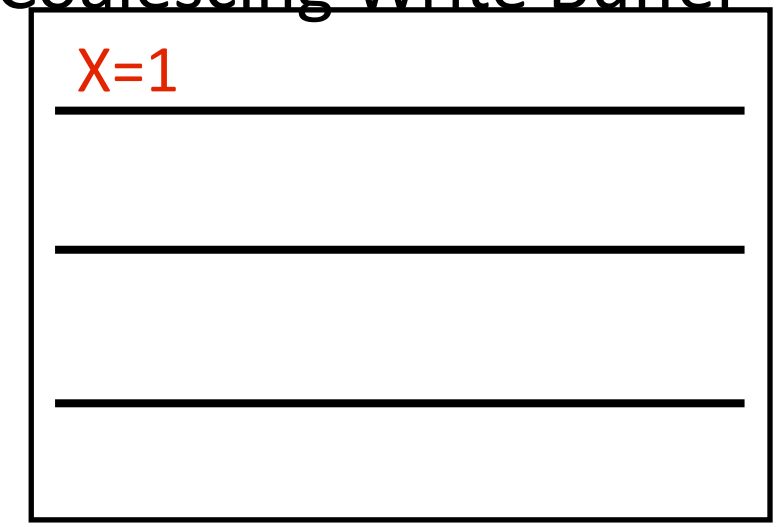
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

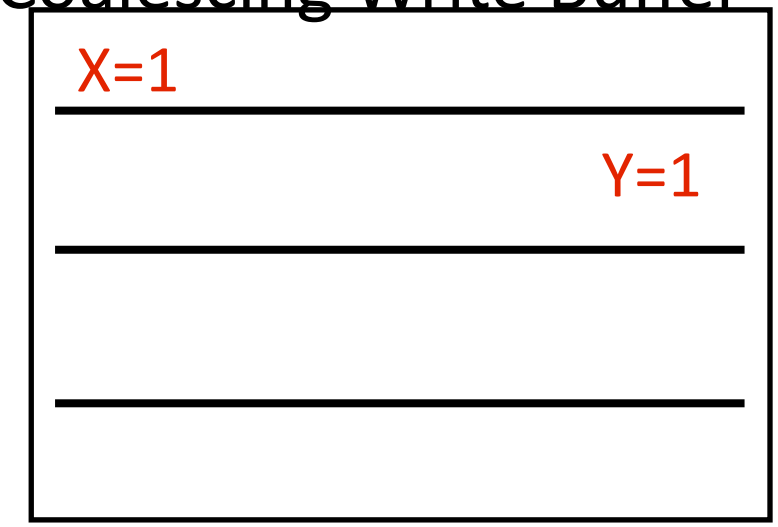
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

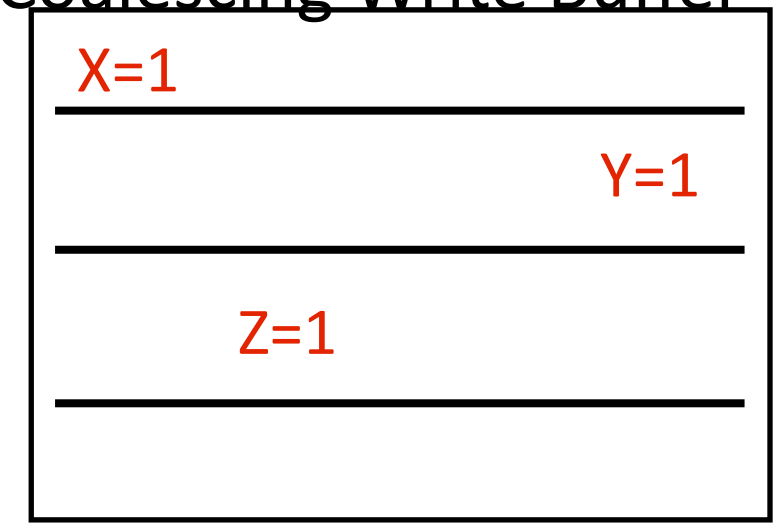
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

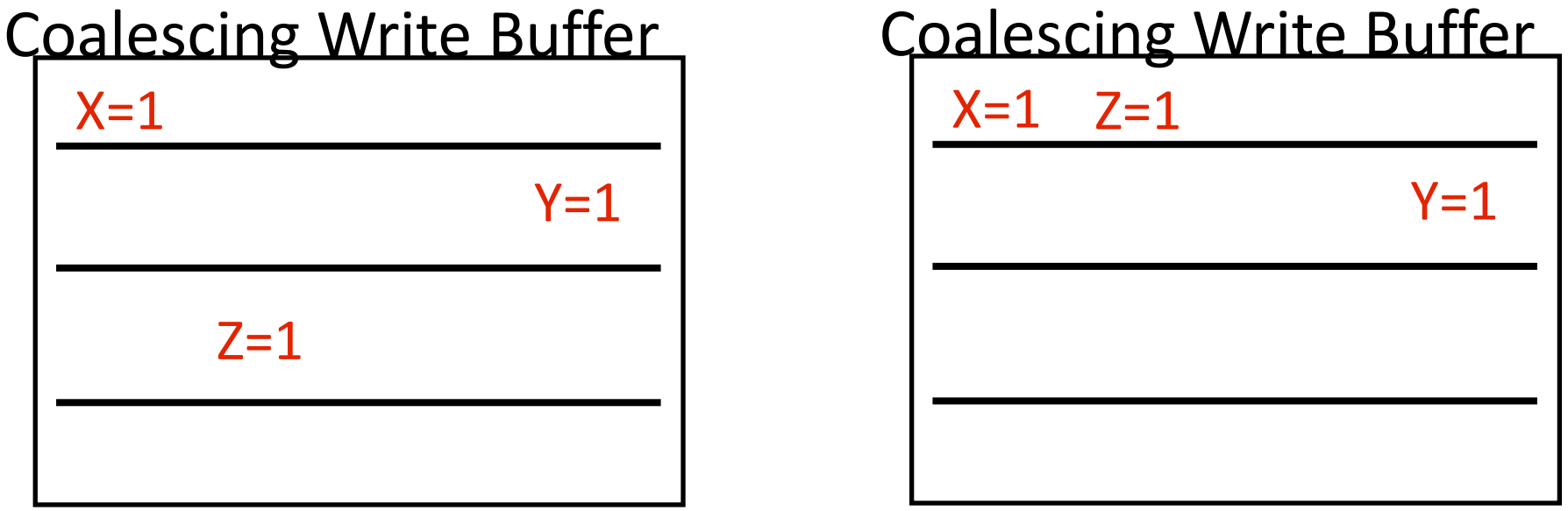
X,Z in same $ line

X=1

Y=1

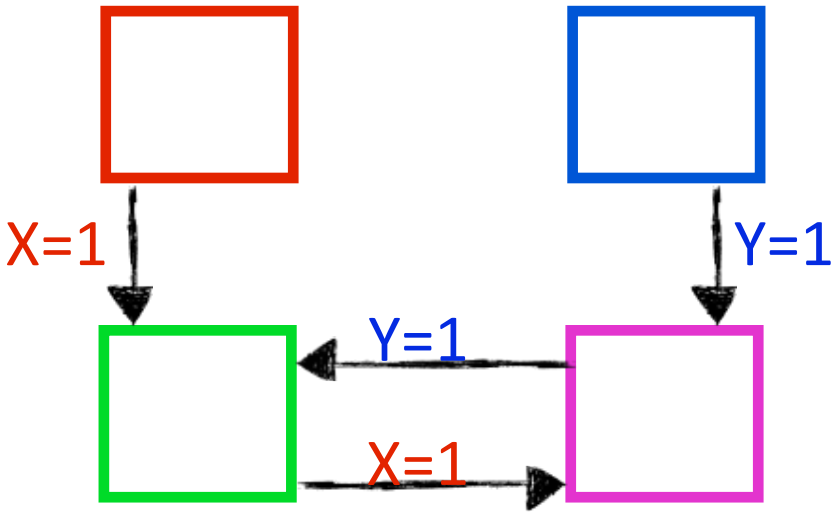
Z=1

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

Program

X=1	r1=X	Y=1	r3=Y
	r2=Y		r4=X

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
```

Hoisted!

```
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 Reordering a Problem?

When Executions Aren’t SC

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

X=1 Y=1

r1=Y r2=X

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

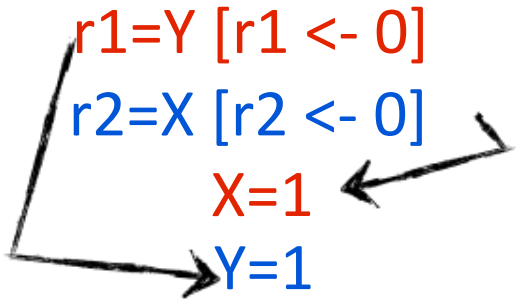
Happens-Before Graph

Program Order HB Edge

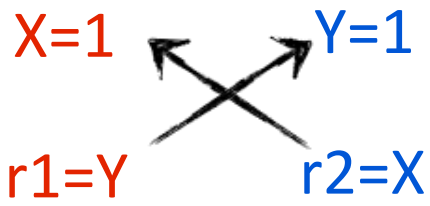
When is an Execution Not SC?

When a memory operation happens before itself

Execution



Happens-Before Graph



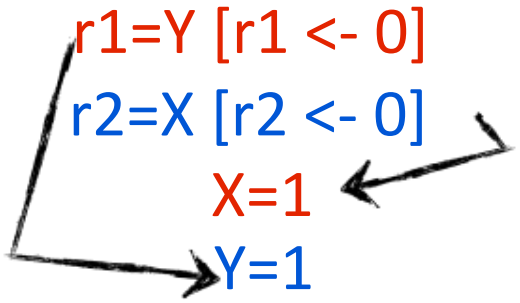
Program Order HB Edge

Causal Order HB Edge

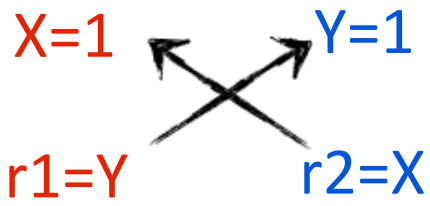
When is an Execution Not SC?

When a memory operation happens before itself

Execution



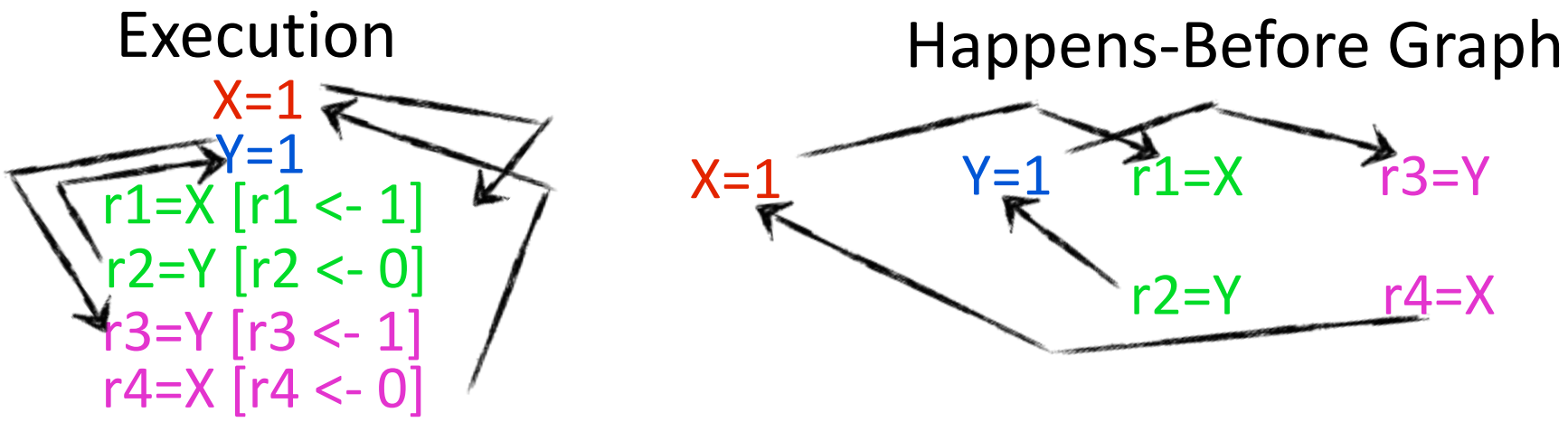
Happens-Before Graph



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

So... are Computers Wrong?!

SC is how **programmers** think.

SC prohibits **all** reordering of instructions

WBs let reads finish before older writes

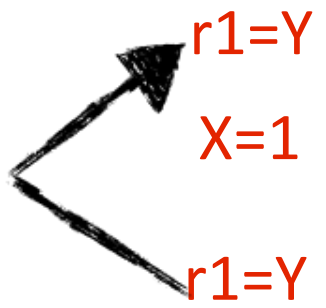
Combining writes saves bandwidth but reorders writes

Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC

x86-TSO (intel x86s)

"The Write Buffer Memory Model"

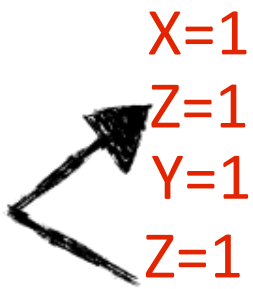


Relaxes W->R order

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

PSO(SPARC)

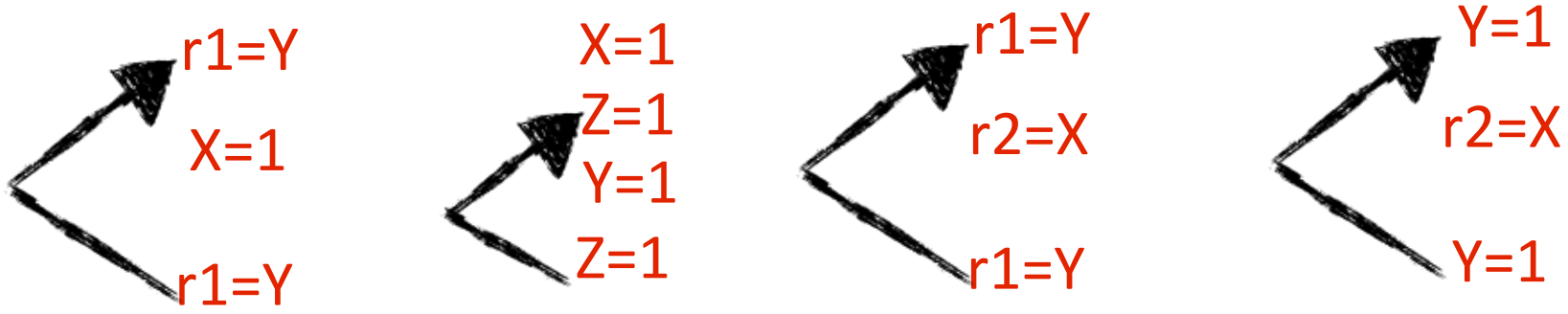
"The Write Combining Memory Model"



Relaxes W->W order

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

In General



W->R W->W R->R R->W

Starting with PSO and relaxing R->R and R->W yields Weak Ordering or Release Consistency (alpha)

Depending on the implementation

SC and Relaxed Consistency

SC is required for correctness and programmer sanity

+

Reordering is required* for performance

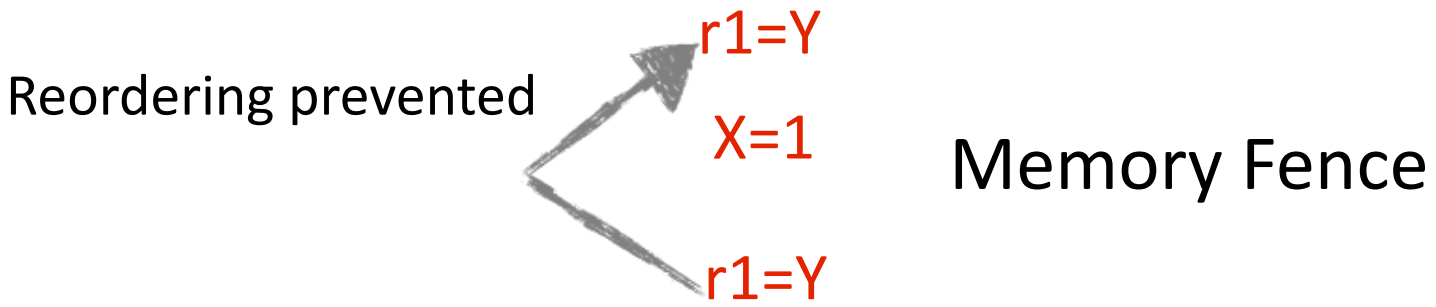
Goal: Ensure SC executions while permitting Relaxed Consistency reorderings

*Usually; the MIPS memory model is **SC** (surprising!)

How to ensure SC, but permit reordering?

Synchronization Prevents Reordering

Memory fences are another type of synchronization



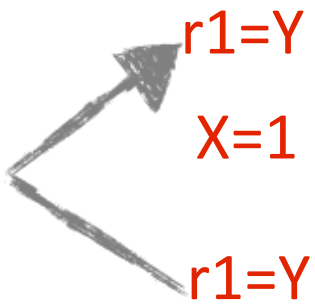
Fence implementation depends on reordering implementation

TSO: Stall reads until write buffer is empty

Synchronization For Real Programmers

Memory fences are wrapped up in locks, etc.

Reordering prevented



Lock

Unlock

Direct use of fences possible, but inadvisable.
USE A LIBRARY.

Data Races

Synchronization imposes happens-before on otherwise unordered operations

Lock

Y=1

Unlock

Lock

r1=Y

Unlock

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

Memory Models across the System Stack

Language

Java/C++: SC for data-race-free programs

Compiler

Conservative with reordering when d-r-f can't be proved

Architecture

Usually very weak for max optimization (lots of reordering)

Note: fences from "above" ensure SC