Lecture 11:

Cache Coherence

Parallel Computing
Stanford CS149, Fall 2023

in-memory, fault-tolerant distributed computing

http://spark.apache.org/

[Zaharia et al. NSDI 2012]

Goals

- Programming model for cluster-scale computations where there is significant reuse of intermediate datasets
  - Iterative machine learning and graph algorithms
  - Interactive data mining: load large dataset into aggregate memory of cluster and then perform multiple ad-hoc queries

- Don't want incur inefficiency of writing intermediates to persistent distributed file system (want to keep it in memory)
  - Challenge: efficiently implementing fault tolerance for large-scale distributed in-memory computations

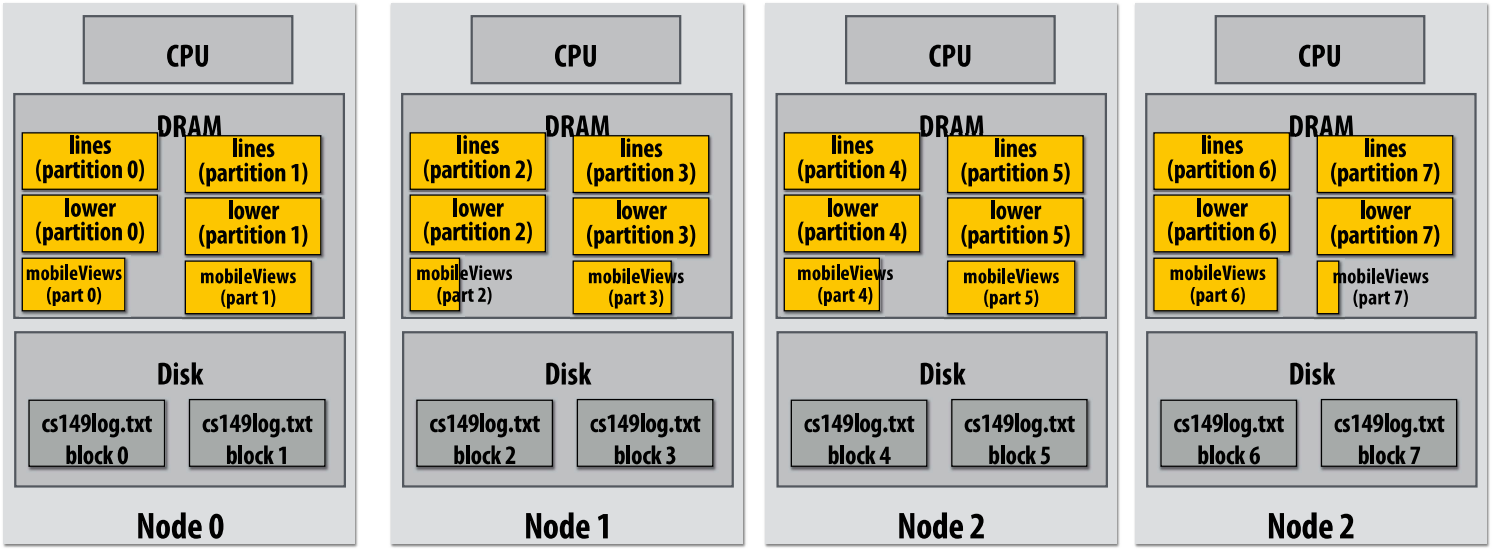
How do we implement RDDs?

In particular, how should they be stored?

Parallel Performance = Parallelism + Locality

```
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

In-memory representation would be huge! (larger than original file on disk)



Resilient Distributed Dataset (RDD)

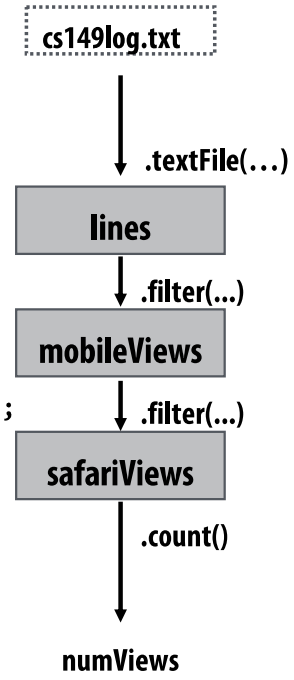
Spark's key programming abstraction:

- Read-only ordered collection of records (immutable)
- RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs
- Actions on RDDs return data to application

RDDs

```
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
val mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
val safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
val numViews = safariViews.count();
```

int



Review: which program performs better?

Program 1

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Program 2

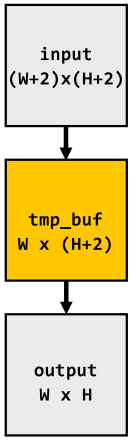
```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
        E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Four loads, one store per 3 math ops (arithmetic intensity = 3/5)

The transformation of the code in program 1 to the code in program 2 is called "loop fusion"

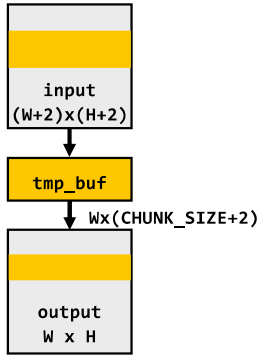
Review: why did we perform this transform?

Program 1




```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.0/3, 1.0/3, 1.0/3};

// blur image horizontally
for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

// blur tmp_buf vertically
for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

Program 2




```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.0/3, 1.0/3, 1.0/3};

for (int j=0; j<HEIGHT; j+CHUNK_SIZE) {

  // blur region of image horizontally
  for (int j2=0; j2<CHUNK_SIZE+2; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+j2)*(WIDTH+2) + i+ii] *
weights[ii];
      tmp_buf[j2*WIDTH + i] = tmp;

  // blur tmp_buf vertically
  for (int j2=0; j2<CHUNK_SIZE; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<3; jj++)
        tmp += tmp_buf[(j2+jj)*WIDTH + i] * weights[jj];
      output[(j+j2)*WIDTH + i] = tmp;
    }
}
```

Both of the previous examples involved globally restructuring the order of computation to improve producer-consumer locality

(improve arithmetic intensity of program)

Fusion with RDDs

- Why is it possible to fuse RDD transformations such as map and filter but not possible with transformations such as groupByKey and Sort?

Implementing sequence of RDD ops efficiently

```
val lines = spark.textFile(“hdfs://cs149log.txt”);
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

Recall “loop fusion” examples

The following code stores only a line of the log file in memory, and only reads input data from disk once (“streaming” solution)

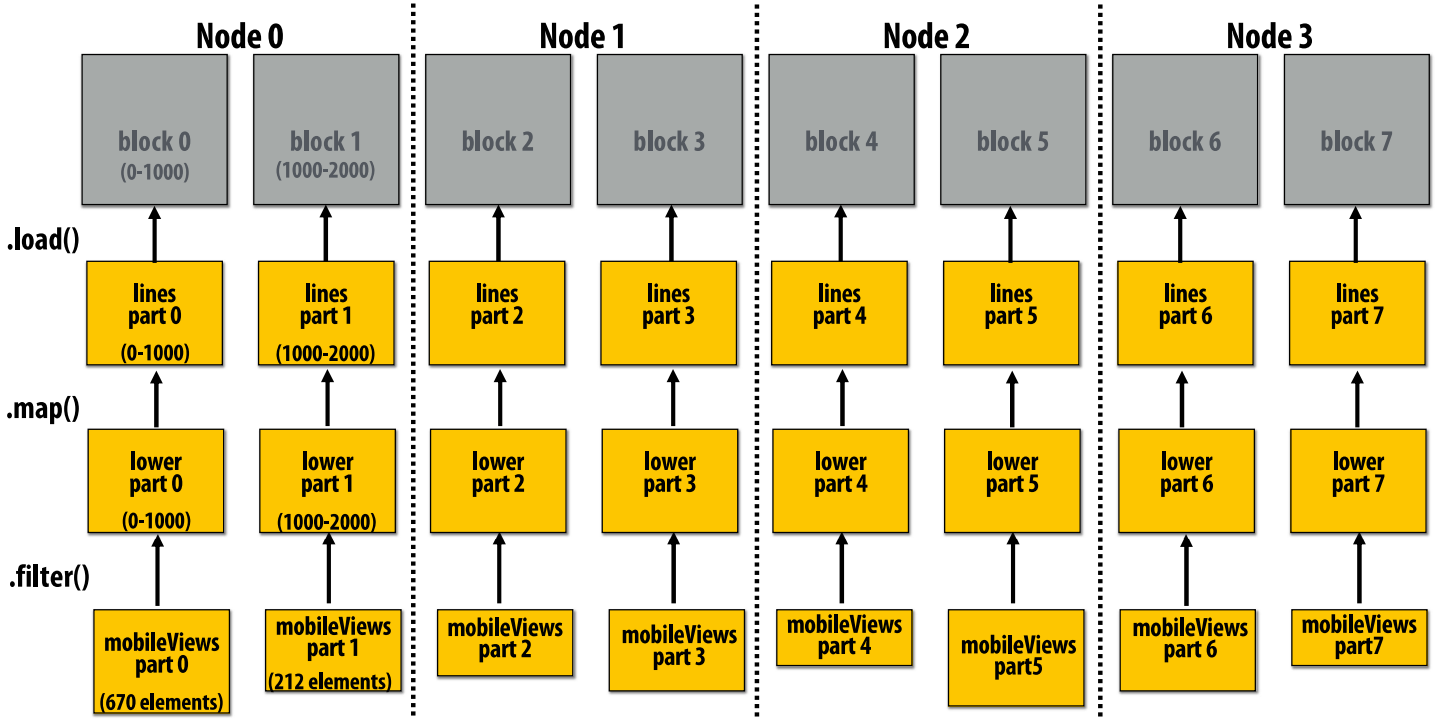
```
int count = 0;
while (inputFile.eof()) {
   string line = inputFile.readLine();
   string lower = line.toLower;
   if (isMobileClient(lower))
     count++;
}
```

Narrow dependencies

```
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

"Narrow dependencies" = each partition of parent RDD referenced by at most one child RDD partition

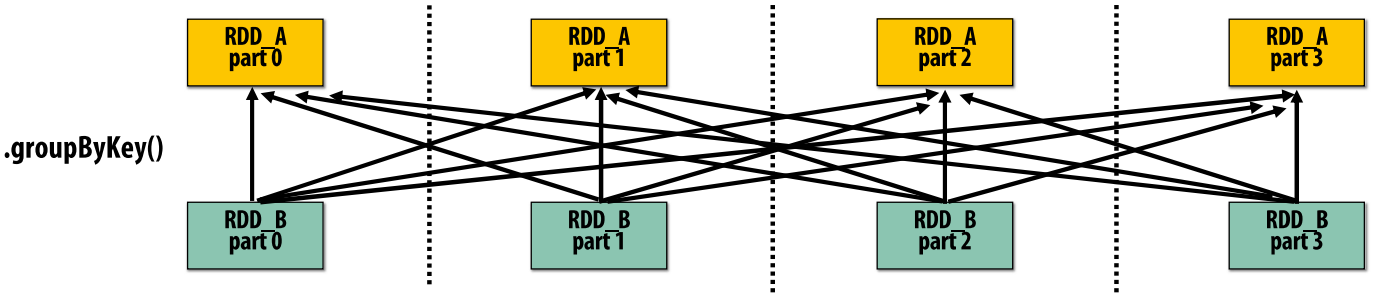
- Allows for fusing of operations (here: can apply map and then filter all at once on input element)
- In this example: no communication between nodes of cluster for transfromations (communication of one int at end to perform count() reduction)



Wide dependencies

groupByKey: RDD[(K,V)] ⟶ RDD[(K,Seq[V])]

"Make a new RDD where each element is a sequence containing all values from the parent RDD with the same key."



Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions

Challenges:

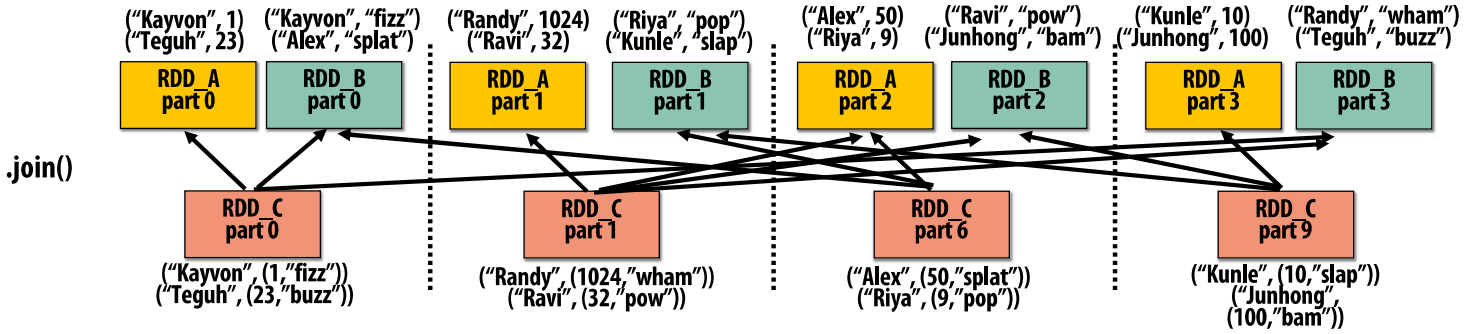
- Must compute all of RDD_A before computing RDD_B
  - Example: groupByKey() may induce all-to-all communication as shown above
- May trigger significant recomputation of ancestor lineage upon node failure
  (I will address resilience in a few slides)

Cost of operations depends on partitioning

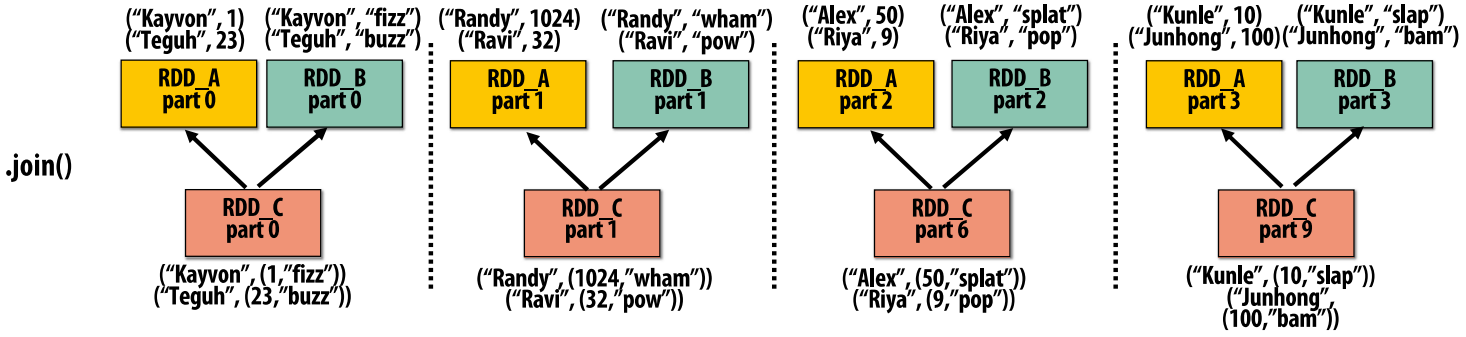
join: RDD[(K,V)], RDD[(K,W)] → RDD[(K,(V,W))]

Assume data in RDD_A and RDD_B are partitioned by key: hash username to partition id

RDD_A and RDD_B have different hash partitions: join creates wide dependencies



RDD_A and RDD_B have same hash partition: join only creates narrow dependencies



PartitionBy() transformation

- Inform Spark on how to partition an RDD
  - e.g., HashPartitioner, RangePartitioner

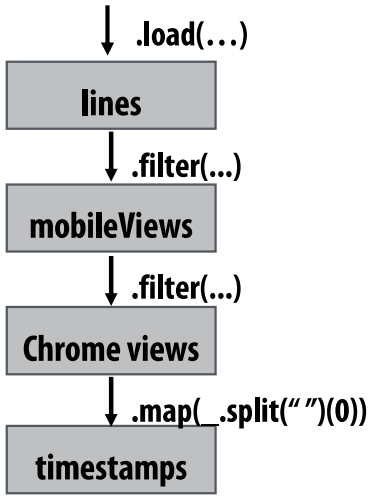
```
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");
val clientInfo = spark.textFile("hdfs://clientssupported.txt"); // (useragent, "yes"/"no")

// create RDD using filter() transformation on lines
val mobileViews = lines.filter(x => isMobileClient(x)).map(x => parseUserAgent(x));

// HashPartitioner maps keys to integers
val partitioner = spark.HashPartitioner(100);

// inform Spark of partition
// .persist() also instructs Spark to try to keep dataset in memory
val mobileViewPartitioned = mobileViews.partitionBy(partitioner)
                                       .persist();
val clientInfoPartitioned = clientInfo.partitionBy(partitioner)
                                       .persist();

// join useragents with whether they are supported or not supported
// Note: this join only creates narrow dependencies due to the explicit partitioning above
void joined = mobileViewPartitioned.join(clientInfoPartitioned);
```

- **.persist():**
  - **Inform Spark this RDD's contents should be retained in memory**
  - **.persist(RELIABLE) = store contents in durable storage (like a checkpoint)**

Implementing Resilience via Lineage

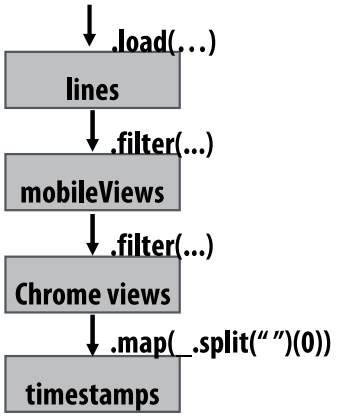
- RDD transformations are bulk, deterministic, and functional
  - Implication: runtime can always reconstruct contents of RDD from its lineage (the sequence of transformations used to create it)
  - Lineage is a log of transformations
  - Efficient: since the log records bulk data-parallel operations, overhead of logging is low (compared to logging fine-grained operations, like in a database)

```
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
val mobileViews = lines.filter((x: String) => isMobileClient(x));

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of
//    page view (first element)
// 3. convert RDD To a scalar sequence (collect() action)
val timestamps = mobileView.filter(_.contains("Chrome"))
                           .map(_.split(" ")(0));
```

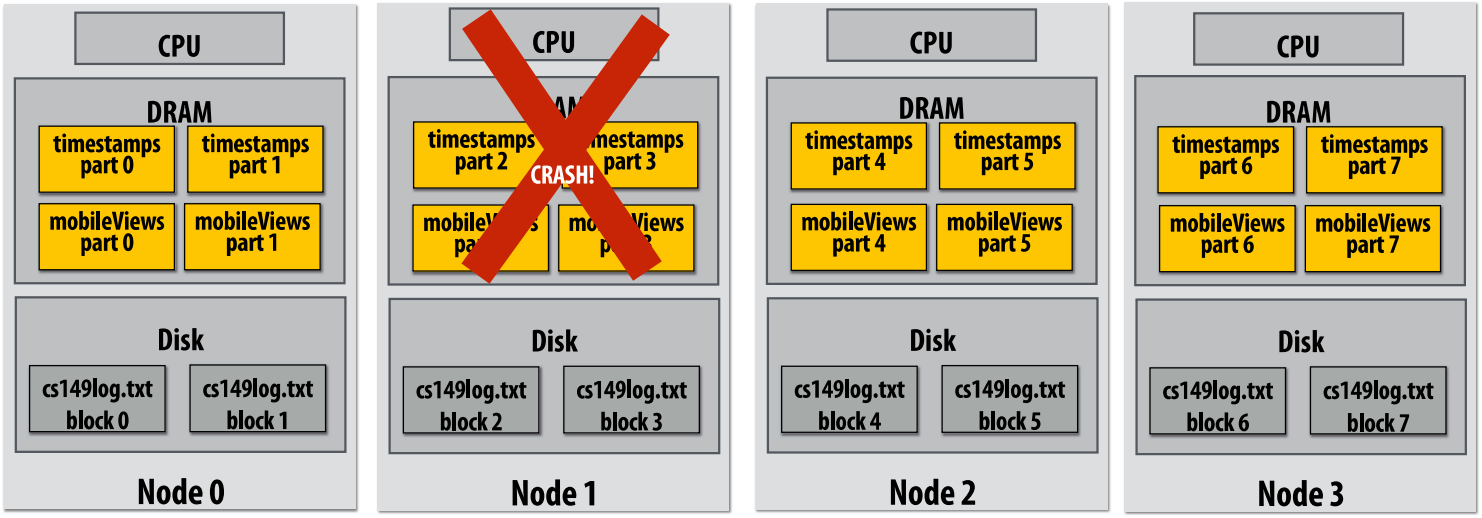


Upon Node Failure: Recompute Lost RDD Partitions from Lineage

```
val lines      = spark.textFile("hdfs://cs149log.txt");
val mobileViews = lines.filter((x: String) => isMobileClient(x));
val timestamps  = mobileView.filter(_.contains("Chrome"))
                            .map(_.split(" ")(0));
```



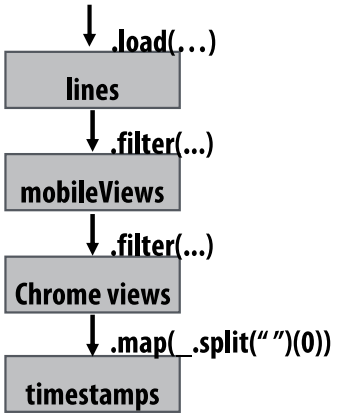
Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.



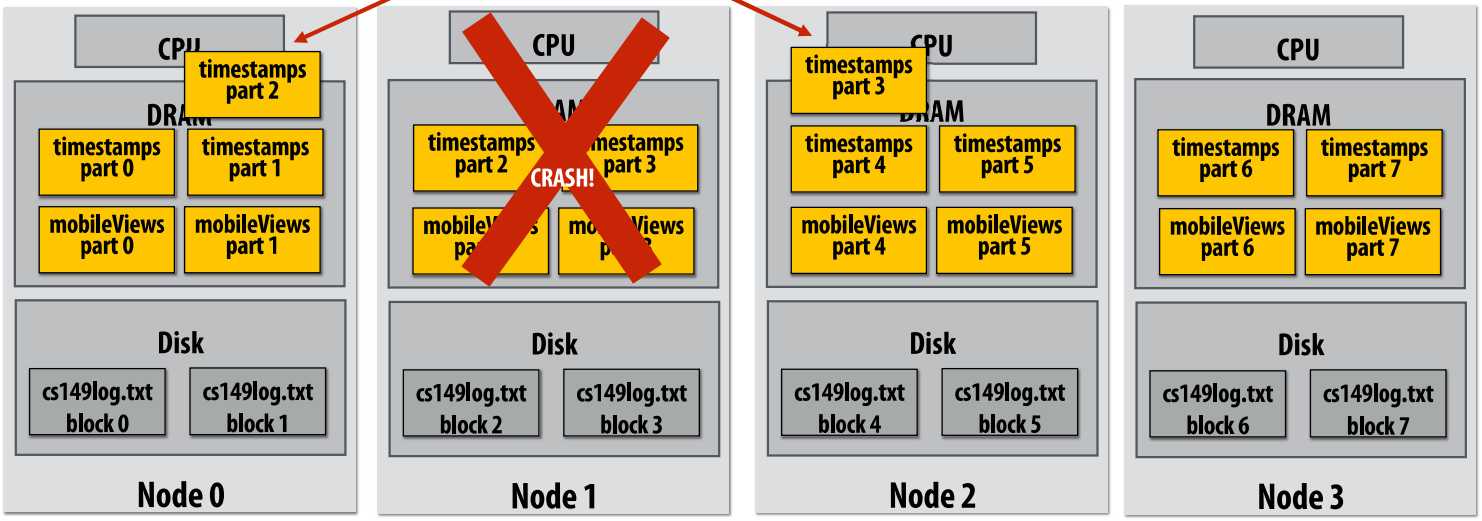
Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes

Upon Node Failure: Recompute Lost RDD Partitions from Lineage

```
val lines      = spark.textFile("hdfs://cs149log.txt");
val mobileViews = lines.filter((x: String) => isMobileClient(x));
val timestamps  = mobileView.filter(_.contains("Chrome"))
                            .map(_.split(" ")(0));
```

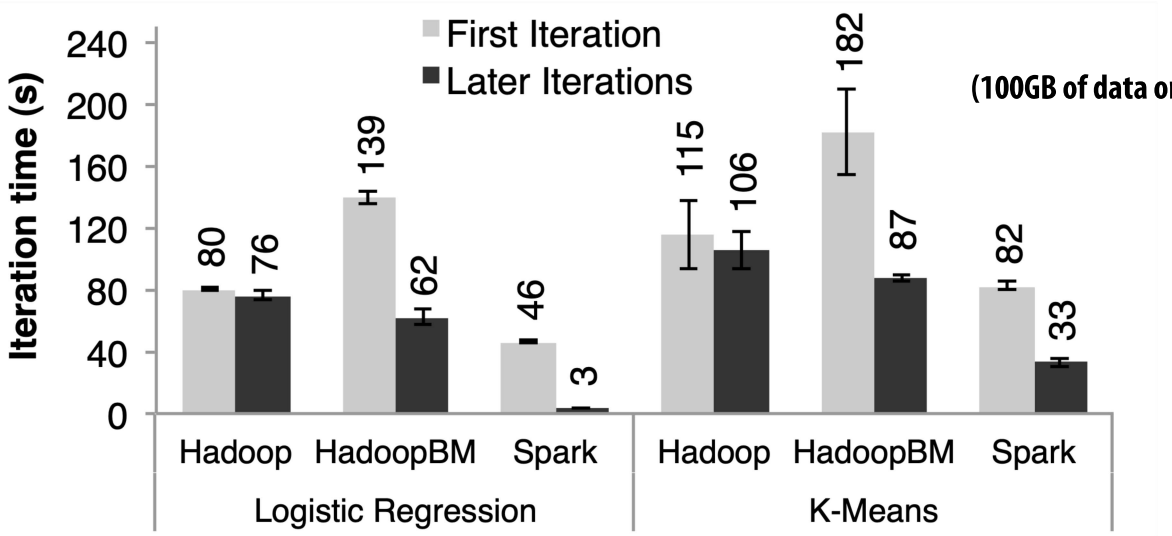


Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps



Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes

Spark performance



(100GB of data on a 100 node cluster)

HadoopBM = Hadoop Binary In-Memory (convert text input to binary, store in in-memory version of HDFS)

Anything else puzzling here?

Q. Wait, the baseline parses text input in each iteration of an iterative algorithm?
A. Yes.

HadoopBM's first iteration is slow because it runs an extra Hadoop job to copy binary form of input data to in memory HDFS

Accessing data from HDFS, even if in memory, has high overhead:

- Multiple mem copies in file system + a checksum
- Conversion from serialized form to Java object

Modern Spark ecosystem

Compelling feature: enables integration/composition of multiple domain-specific frameworks (since all collections implemented under the hood with RDDs and scheduled using Spark scheduler)


```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Interleave computation and database query
Can apply transformations to RDDs produced by SQL queries

Machine learning library build on top of Spark abstractions.

```
points = spark.textFile("hdfs://...")
              .map(parsePoint)

model = KMeans.train(points, k=10)
```

GraphLab-like library built on top of Spark abstractions.

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Spark summary

- Introduces opaque sequence abstraction (RDD) to encapsulate intermediates of cluster computations (previously… frameworks like Hadoop/MapReduce stored intermediates in the file system)
  - Observation: "files are a poor abstraction for intermediate variables in large-scale data-parallel programs"
  - RDDs are read-only, and created by deterministic data-parallel operators
  - Lineage tracked and used for locality-aware scheduling and fault-tolerance (allows recomputation of partitions of RDD on failure, rather than restore from checkpoint *)
    - Bulk operations allow overhead of lineage tracking (logging) to be low.
- Simple, versatile abstraction upon which many domain-specific distributed computing frameworks are being implemented.
  - See Apache Spark project: spark.apache.org

*** Note that .persist(RELIABLE) allows programmer to request checkpointing in long lineage situations.**

Caution: "scale out" is not the entire story

- Distributed systems designed for cloud execution address many difficult challenges, and have been instrumental in the explosion of "big-data" computing and large-scale analytics
  - Scale-out parallelism to many machines
  - Resiliency in the face of failures
  - Complexity of managing clusters of machines
- But scale out is not the whole story:

20 Iterations of Page Rank

scalable system	cores	twitter	uk-2007-05
GraphChi [10]	2	3160s	6972s
Stratosphere [6]	16	2250s	-
X-Stream [17]	16	1488s	-
Spark [8]	128	857s	1759s
Giraph [8]	128	596s	1235s
GraphLab [8]	128	249s	833s
GraphX [8]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

name	twitter_rv [11]	uk-2007-05 [4]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

Vertex order (SSD)	1	300s	651s
Vertex order (RAM)	1	275s	-
Hilbert order (SSD)	1	242s	256s
Hilbert order (RAM)	1	110s	-

Further optimization of the baseline brought time down to 110s

["Scalability! At what COST?" McSherry et al. HotOS 2015]

Caution: "Scale Out" is Not the Entire Story

Label Propagation
[McSherry et al. HotOS 2015]

scalable system	cores	twitter	uk-2007-05
Stratosphere [6]	16	950s	-
X-Stream [17]	16	1159s	-
Spark [8]	128	1784s	$\geq$ 8000s
Giraph [8]	128	200s	$\geq$ 8000s
GraphLab [8]	128	242s	714s
GraphX [8]	128	251s	800s
Single thread (SSD)	1	153s	417s

from McSherry 2015:

"The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform. While nearly all such publications detail their system's impressive scalability, few directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?"

COST = "Configuration that Outperforms a Single Thread"

Perhaps surprisingly, many published systems have unbounded COST—i.e., no configuration outperforms the best single-threaded implementation—for all of the problems to which they have been applied.

BID Data Suite (1 GPU accelerated node)
[Canny and Zhao, KDD 13]

Page Rank

System	Graph VxE	Time(s)	Gflops	Procs
Hadoop	?x1.1B	198	0.015	50x8
Spark	40Mx1.5B	97.4	0.03	50x2
Twister	50Mx1.4B	36	0.09	60x4
PowerGraph	40Mx1.4B	3.6	0.8	64x8
BIDMat	60Mx1.4B	6	0.5	1x8
BIDMat+disk	60Mx1.4B	24	0.16	1x8

Latency Dirichlet Allocation (LDA)

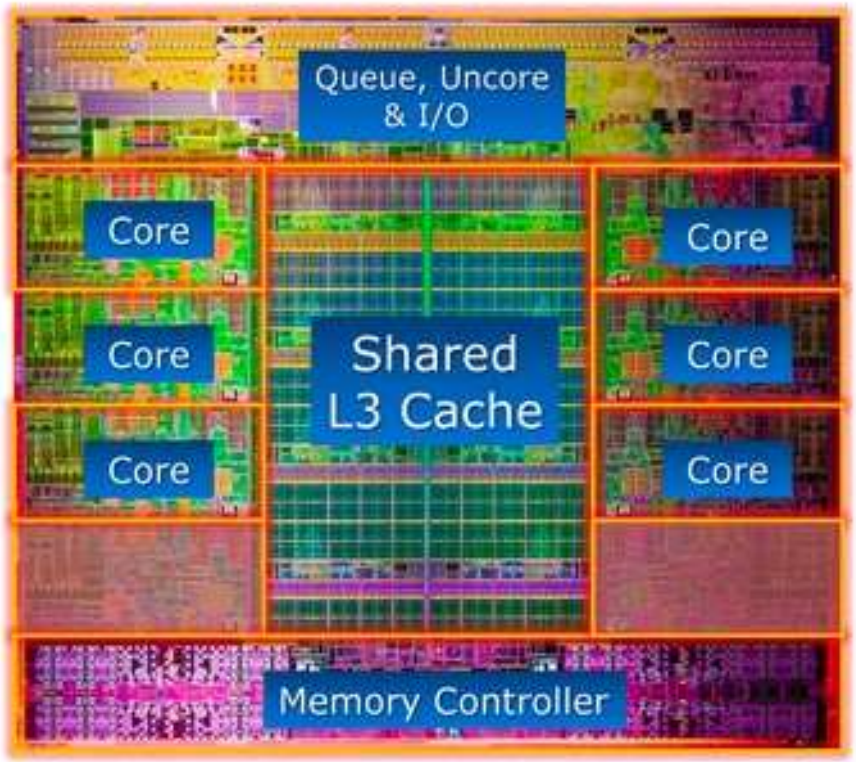
System	Docs/hr	Gflops	Procs
Smola[15]	1.6M	0.5	100x8
PowerGraph	1.1M	0.3	64x16
BIDMach	3.6M	30	1x8x1

Lecture 10:

Cache Coherence

Parallel Computing
Stanford CS149, Fall 2023

Intel Core i7



- 30% of the die area is cache

Rewview:Cache example 1

Array of 16 bytes in memory

	Address	Value
Line 0x0	0x0	16
	0x1	255
	0x2	14
	0x3	0
Line 0x4	0x4	0
	0x5	0
	0x6	6
	0x7	0
Line 0x8	0x8	32
	0x9	48
	0xA	255
	0xB	255
Line 0xC	0xC	255
	0xD	0
	0xE	0
	0xF	0

Assume:

Total cache capacity of 8 bytes

Cache with 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU)
replacement policy

time

Address accessed	Cache action	Cache state (after load is complete)
0x0	"cold miss", load 0x0	0x0
0x1	hit	0x0
0x2	hit	0x0
0x3	hit	0x0
0x2	hit	0x0
0x1	hit	0x0
0x4	"cold miss", load 0x4	0x0, 0x4
0x1	hit	0x0, 0x4

There are two forms of "data locality" in this sequence:

Spatial locality: loading data in a cache line "preloads" the data needed for subsequent accesses to different addresses in the same line, leading to cache hits

Temporal locality: repeated accesses to the same address result in hits.

Review: Cache example 2

Array of 16 bytes in memory

	Address	Value
Line 0x0	0x0	16
	0x1	255
	0x2	14
	0x3	0
Line 0x4	0x4	0
	0x5	0
	0x6	6
	0x7	0
Line 0x8	0x8	32
	0x9	48
	0xA	255
	0xB	255
Line 0xC	0xC	255
	0xD	0
	0xE	0
	0xF	0

Assume:

Total cache capacity of 8 bytes

Cache with 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU)
replacement policy

time

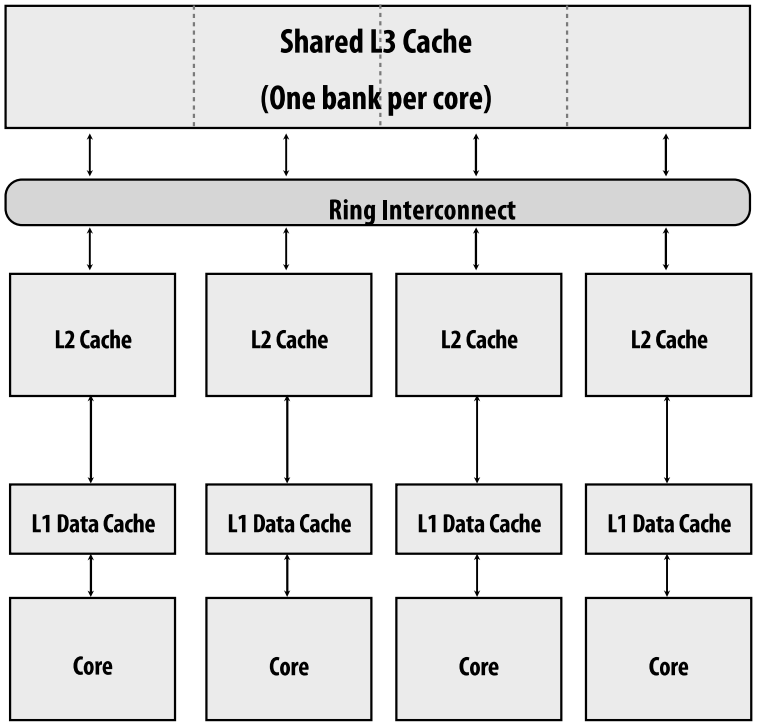
Address accessed	Cache action	Cache state (after load is complete)
0x0	"cold miss", load 0x0	0x0
0x1	hit	0x0
0x2	hit	0x0
0x3	hit	0x0
0x4	"cold miss", load 0x4	0x0 0x4
0x5	hit	0x0 0x4
0x6	hit	0x0 0x4
0x7	hit	0x0 0x4
0x8	"cold miss", load 0x8 (evict 0x0)	0x8 0x4
0x9	hit	0x8 0x4
0xA	hit	0x8 0x4
0xB	hit	0x8 0x4
0xC	"cold miss", load 0xC (evict 0x4)	0x8 0xC
0xD	hit	0x8 0xC
0xE	hit	0x8 0xC
0xF	hit	0x8 0xC
0x0	"capacity miss", load 0x0 (evict 0x8)	0x0 0xC

Cache hierarchy of Intel Skylake CPU (2015)

Caches exploit locality

3 Cs cache miss model

- Cold
- Capacity
- Conflict



L3: (per chip)
8 MB, inclusive
16-way set associative
32B / clock per bank
42 cycle latency

64 byte cache line size

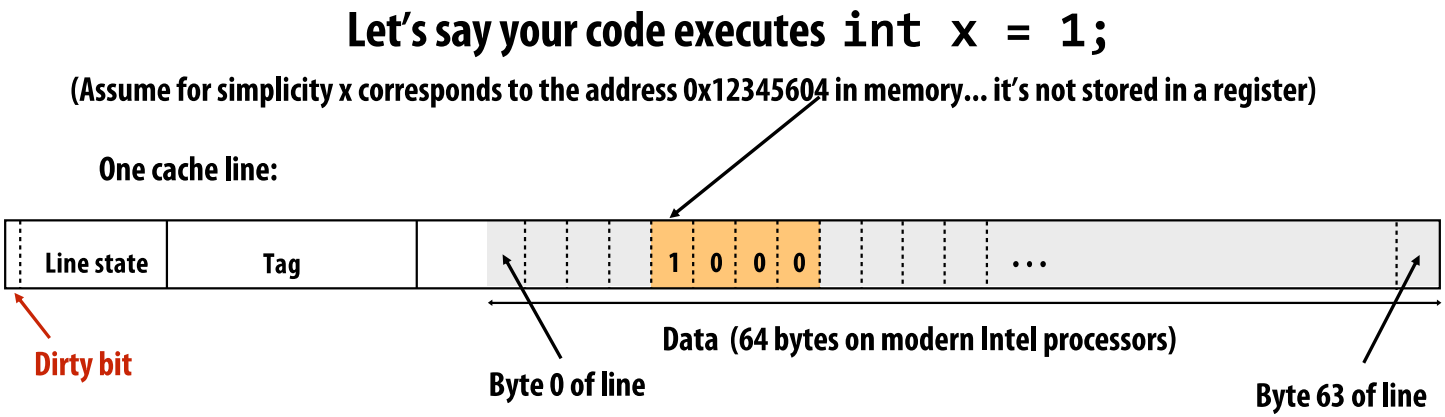
L2: (private per core)
256 KB
4-way set associative, write back
64B / clock, 12 cycle latency

L1: (private per core)
32 KB
8-way set associative, write back
2 x 32B load + 1 x 32B store per clock
4 cycle latency

Support for:
72 outstanding loads
56 outstanding stores

Source: Intel 64 and IA-32 Architectures Optimization Reference Manual (June 2016)

Cache Design



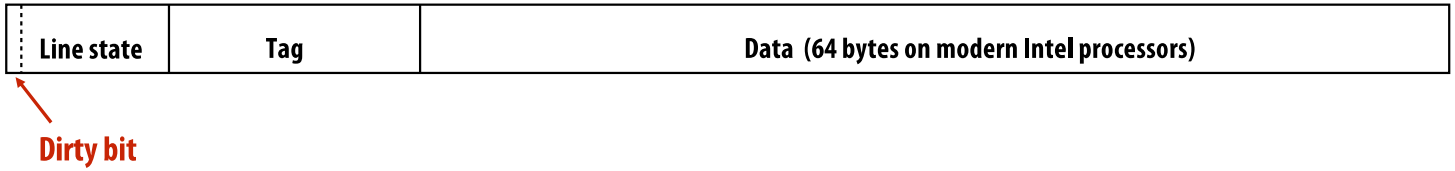
- Do you know the difference between a write back and a write-through cache?

- What about a write-allocate vs. write-no-allocate cache?

Behavior of write-allocate, write-back cache on a write miss (uniprocessor case)

Example: processor executes `int x = 1;`

1. Processor performs write to address that "misses" in cache
2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory
3. Cache loads line from memory ("allocates line in cache")
4. Whole cache line is fetched and 32 bits are updated
5. Cache line is marked as dirty



Review: Shared address space model (abstraction)

- Threads Reading/writing to shared variables
  - Inter-thread communication is implicit in memory operations
  - Thread 1 stores to X
  - Later, thread 2 reads X (and observes update of value by thread 1)

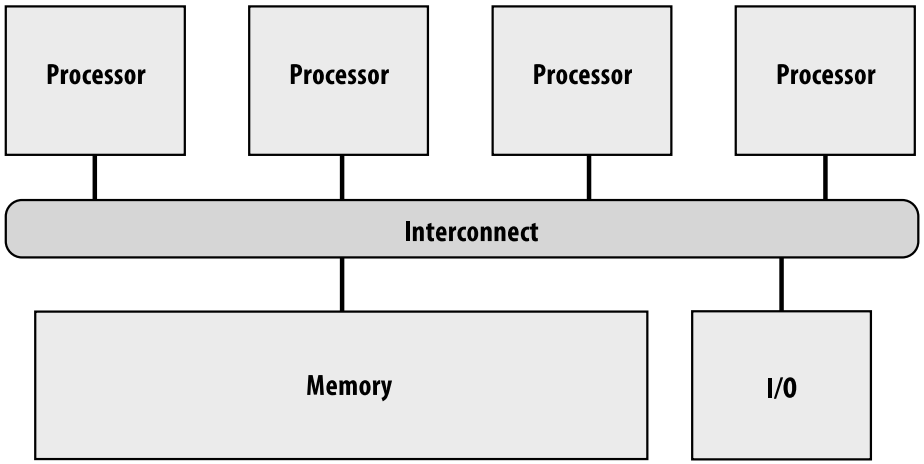
  - Manipulating synchronization primitives
    - e.g., ensuring mutual exclusion via use of locks

- This is a natural extension of sequential programming

A shared memory multi-processor

- Processors read and write to shared variables
  - More precisely: processors issue load and store instructions

- A reasonable expectation of memory is:
  - Reading a value at address X should return the last value written to address X by any processor

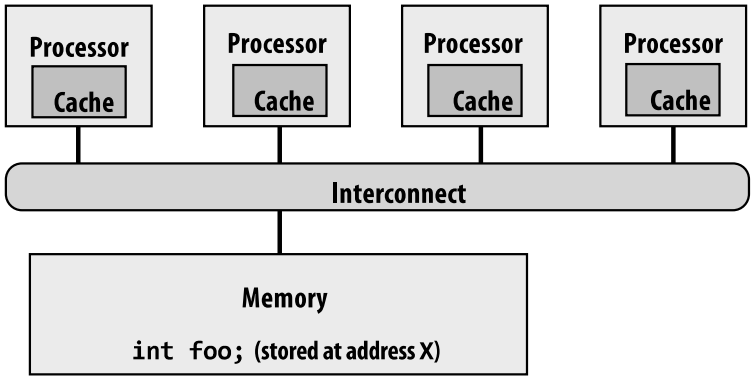


(A simple view of four processors and their shared address space)

The cache coherence problem

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location



The chart at right shows the value of variable `foo` (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

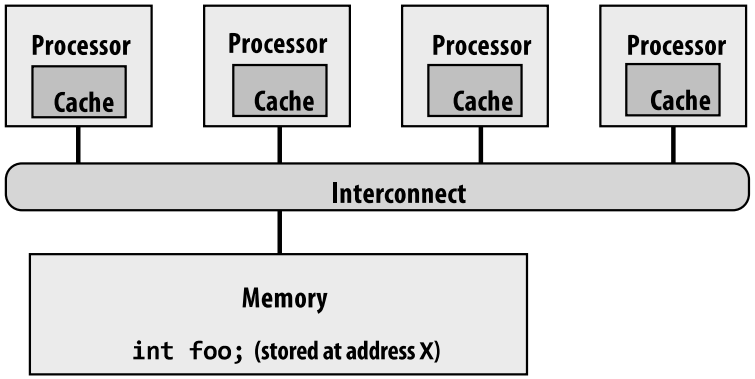
Assume write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

The cache coherence problem

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location



Is this a mutual exclusion problem?

Can you fix the problem by adding locks to your program?

NO!

This is a problem created by replicating the data stored at address X in local caches

The chart at right shows the value of variable `foo` (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

Assume write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

How could we fix this problem?

The memory coherence problem

- Intuitive behavior for memory system: reading value at address X should return the last value written to address X *by any processor*.

- Memory coherence problem exists because there is both global storage (main memory) and per-processor local storage (processor caches) implementing the abstraction of a single shared address space.

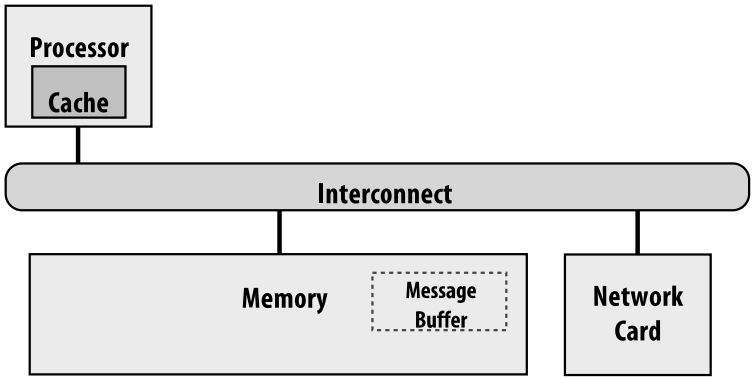
Intuitive expectation of shared memory

- Intuitive behavior for memory system: reading value at address X should return the last value written to address X *by any processor*.

- On a uniprocessor, providing this behavior is fairly simple, since writes typically come from one source: the processor
  - Exception: device I/O via direct memory access (DMA)

Coherence is an issue in a single CPU system

Consider I/O device performing DMA data transfer



Case 1:
Processor writes to buffer in main memory
Processor tells network card to async send buffer
Problem: network card many transfer stale data if processor's writes (reflected in cached copy of data) are not flushed to memory

Case 2:
Network card receives message
Network card copies message in buffer in main memory using DMA transfer
Card notifies CPU msg was received, buffer ready to read
Problem: CPU may read stale data if addresses updated by network card happen to be in cache

- Common solutions:
  - CPU writes to shared buffers using uncached stores (e.g., driver code)
  - OS support:
    - Mark virtual memory pages containing shared buffers as not-cachable
    - Explicitly flush pages from cache when I/O completes

- In practice, DMA transfers are infrequent compared to CPU loads and stores

(so these heavyweight software solutions are acceptable)

Problems with the intuition

- Intuitive behavior: reading value at address X should return the last value written to address X *by any processor*

- What does "last" mean?
  - What if two processors write at the same time?
  - What if a write by P1 is followed by a read from P2 so close in time that it is impossible to communicate the occurrence of the write to P2 in time?

- In a sequential program, "last" is determined by program order (not time)
  - Holds true within one thread of a parallel program
  - But we need to come up with a meaningful way to describe order across threads in a parallel program

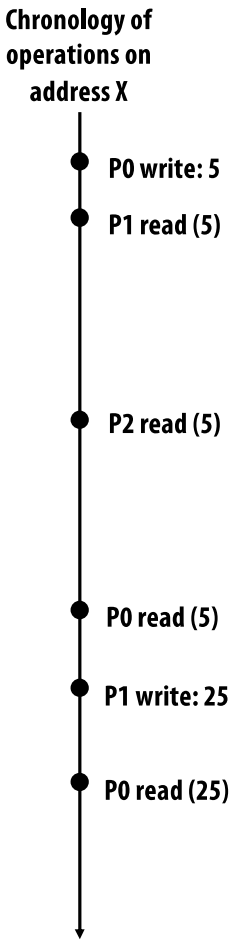
Definition: Coherence

A memory system is coherent if:

The results of a parallel program's execution are such that for each memory location, there is a hypothetical serial order of all program operations (executed by all processors) to the location that is consistent with the results of execution, and:

1. Memory operations issued by any one processor occur in the order issued by the processor

2. The value returned by a read is the value written by the last write to the location . . . as given by the serial order

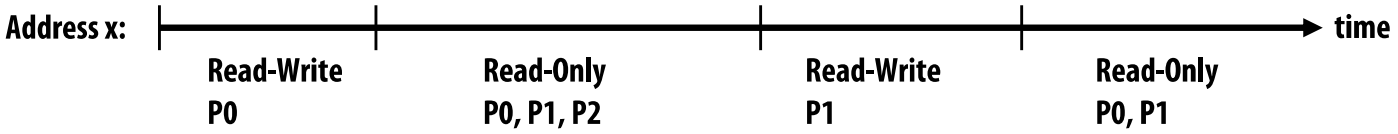


Implementation: Cache Coherence Invariants

For any memory address x, at any given time period (epoch):

- Single-Writer, Multiple-Read (SWMR) Invariant
  - Read-write epoch: there exists only a single processor that may write to x (and can also read it)
  - Read-Only- epoch: some number of processors that may only read x

- Data-Value Invariant (write serialization)
  - The value of the memory address at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch



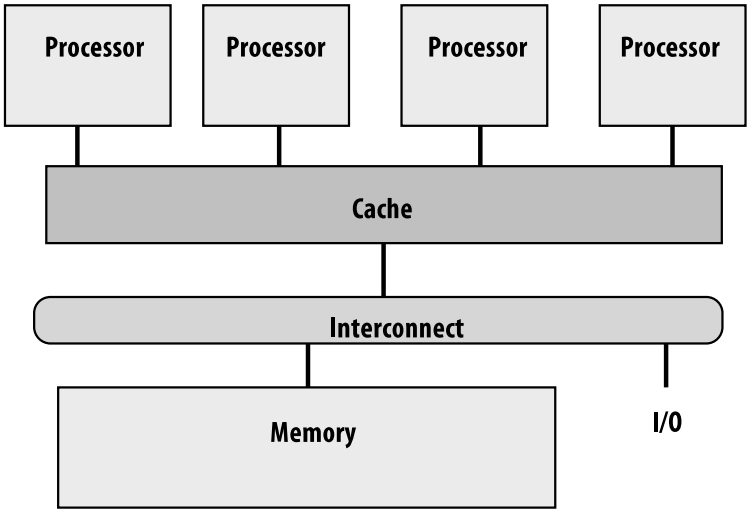
Implementing coherence

- Software-based solutions (coarse grain: VM page)
  - OS uses page-fault mechanism to propagate writes
  - Can be used to implement memory coherence over clusters of workstations
  - We won't discuss these solutions
  - Big performance problem: false sharing (discussed later)

- Hardware-based solutions (fine grain: cache line)
  - "Snooping"-based coherence implementations (today)
  - Directory-based coherence implementations (briefly)

Shared caches: coherence made easy

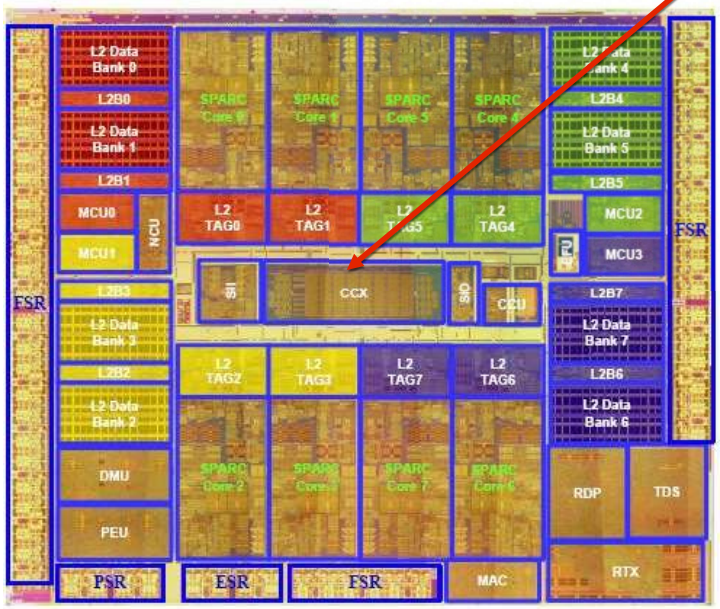
- One single cache shared by all processors
  - Eliminates problem of replicating state in multiple caches
- Obvious scalability problems (since the point of a cache is to be local and fast)
  - Interference (conflict misses) / contention due to many clients (destructive)
- But shared caches can have benefits:
  - Facilitates fine-grained sharing (overlapping working sets)
  - Loads/stores by one processor might pre-fetch lines for another processor (constructive)



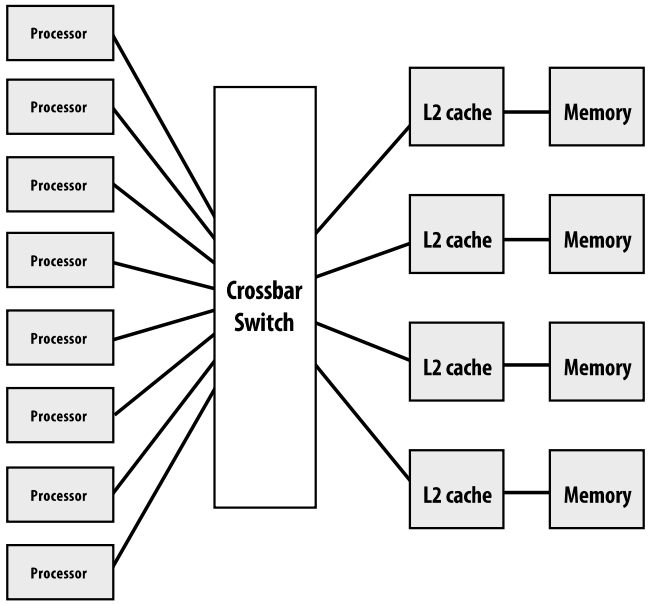

```
forall (i= 0; i++; i< N)
   x[i] = y[i] + y[i+1] + y[i+2];
```

SUN Niagara 2 (UltraSPARC T2)

Note area of crossbar (CCX):
about same area as one core on chip



Eight cores



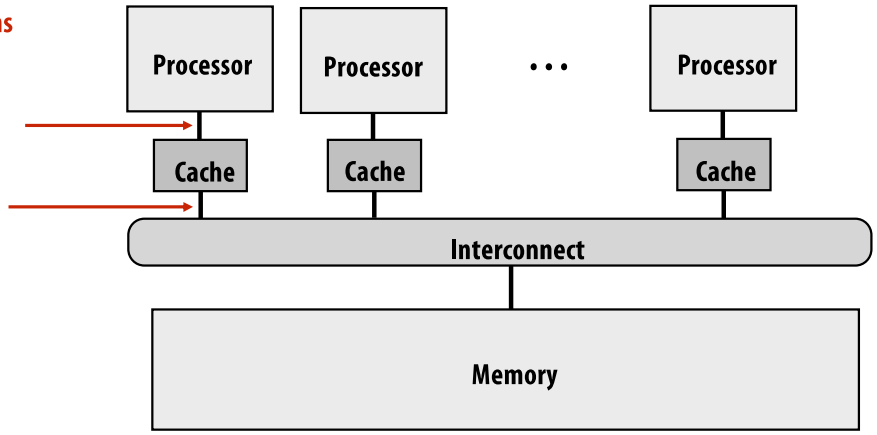
Snooping cache-coherence schemes

- Main idea: all coherence-related activity is broadcast to all processors in the system (more specifically: to the processor's cache controllers)

- Cache controllers monitor ("they snoop") memory operations, and follow cache coherence protocol to maintain memory coherence

Notice: now cache controller must respond to actions from "both ends":

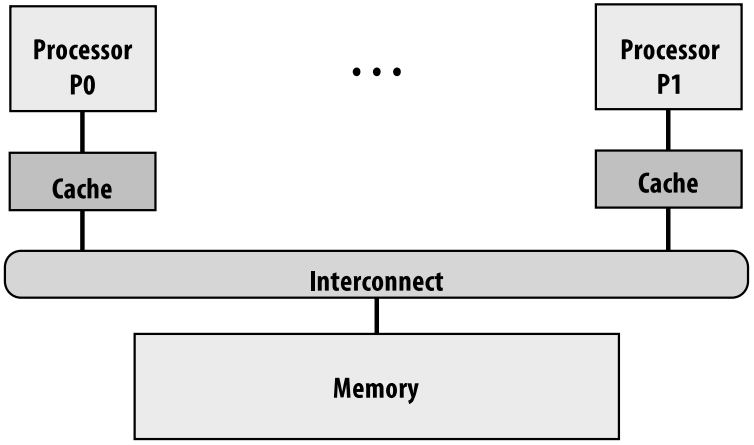
1. LD/ST requests from its local processor
2. Coherence-related activity broadcast over the chip's interconnect



Very simple coherence implementation

Let's assume:

1. Write-through caches
2. Granularity of coherence is cache line



Coherence Protocol:

- Upon write, cache controller broadcasts invalidation message
- As a result, the next read from other processors will trigger cache miss

(processor retrieves updated value from memory due to write-through policy)

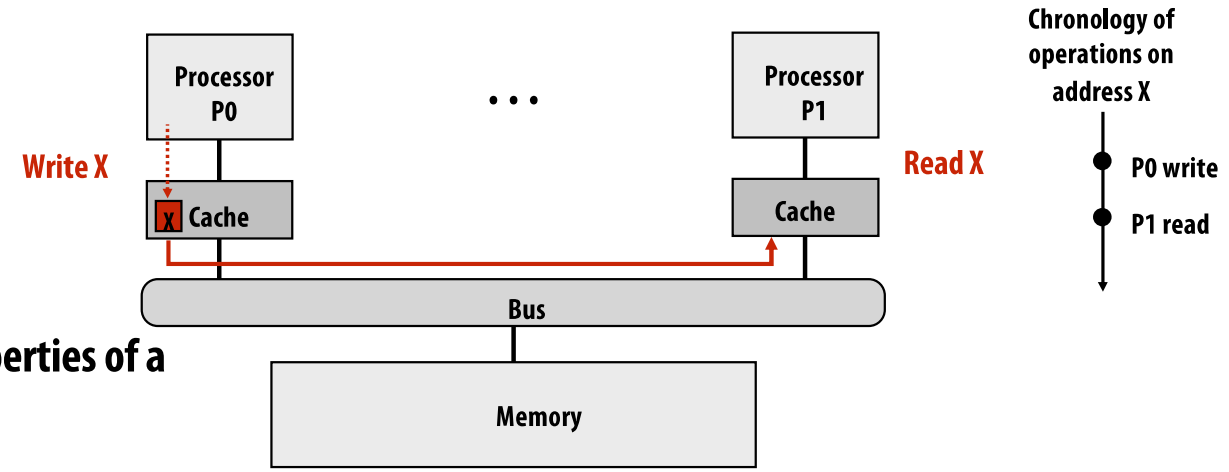
Action	Interconnect activity	P0 $	P1 $	mem location X
				0
P0 load X	cache miss for X	0		0
P1 load X	cache miss for X	0	0	0
P0 write 100 to X	invalidation for X	100		100
P1 load X	cache miss for X	100	100	100

Write-through policy is inefficient

- Every write operation goes out to memory
  - Very high bandwidth requirements

- Write-back caches absorb most write traffic as cache hits
  - Significantly reduces bandwidth requirements
  - But now how do we maintain cache coherence invariants?
  - This requires more sophisticated coherence protocols

Cache coherence with write-back caches



What are two important properties of a bus?

- **Dirty state of cache line now indicates exclusive ownership (Read-Write Epoch)**
  - **Modified: cache is only cache with a valid copy of line (it can safely be written to)**
  - **Owner: cache is responsible for propagating information to other processors when they attempt to load it from memory (otherwise a load from another processor will get stale data from memory)**

Cache Coherence Protocol

- Algorithm that maintains cache coherent invariants

- The logic we are about to describe is performed by each processor's cache controller in response to:
  - Loads and stores by the local processor
  - Messages from other caches on the bus

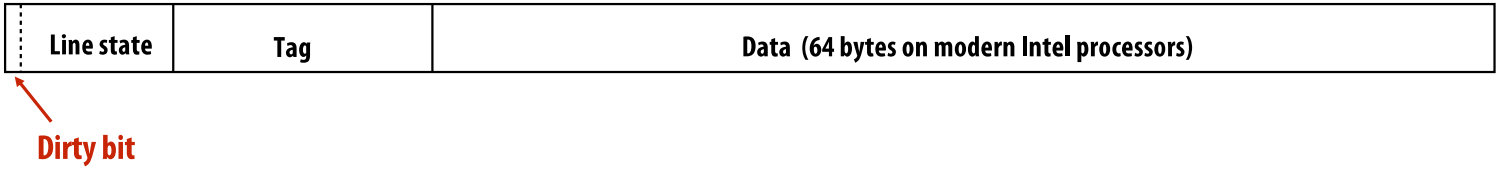
- If all cache controllers operate according to this described protocol, then coherence will be maintained
  - The caches "cooperate" to ensure coherence is maintained

Invalidation-based write-back protocol

Key ideas:

- A line in the "modified" state can be modified without notifying the other caches
- Processor can only write to lines in the modified state
  - Need a way to tell other caches that processor wants exclusive access to the line
  - We accomplish this by sending message to all the other caches
- When cache controller sees a request for modified access to a line it contains
  - It must invalidate the line in its cache

Recall cache line state bits



MSI write-back invalidation protocol

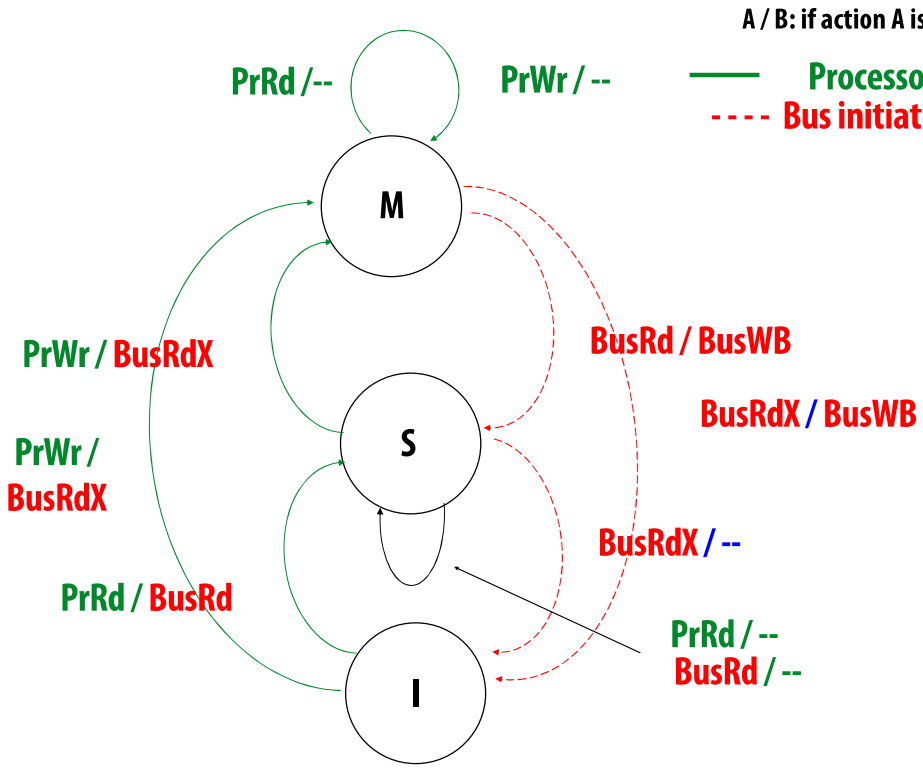
- Key tasks of protocol
  - Ensuring processor obtains exclusive access for a write
  - Locating most recent copy of cache line's data on cache miss

- Three cache line states
  - Invalid (I): same as meaning of invalid in uniprocessor cache
  - Shared (S): line valid in one or more caches, memory is up to date
  - Modified (M): line valid in exactly one cache (a.k.a. "dirty" or "exclusive" state)

- Two processor operations (triggered by local CPU)
  - PrRd (read)
  - PrWr (write)

- Three coherence-related bus transactions (from remote caches)
  - BusRd: obtain copy of line with no intent to modify
  - BusRdX: obtain copy of line with intent to modify
  - BusWB: write dirty line out to memory

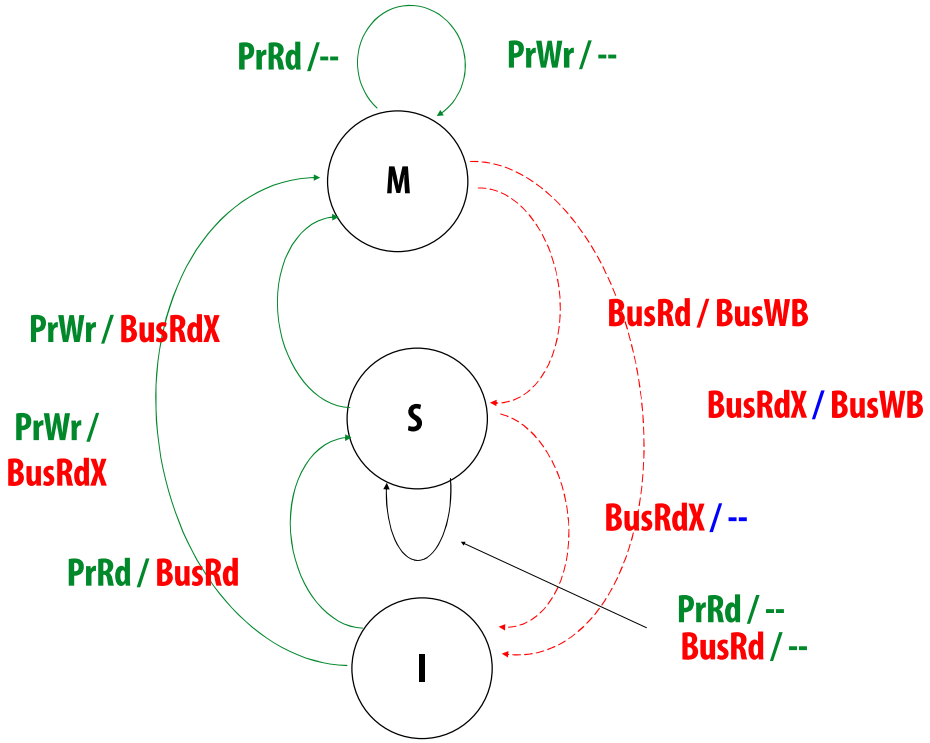
Cache Coherence Protocol: MSI State Diagram



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

MSI Invalidate Protocol

- Read obtains block in "shared"
  - even if only cached copy
- Obtain exclusive ownership before writing
  - BusRdX causes others to invalidate
  - If M in another cache, will cause writeback
  - BusRdX even if hit in S
    - promote to M (upgrade)



* Remember, all caches are carrying out this logic independently to maintain coherence

A Cache Coherence Example

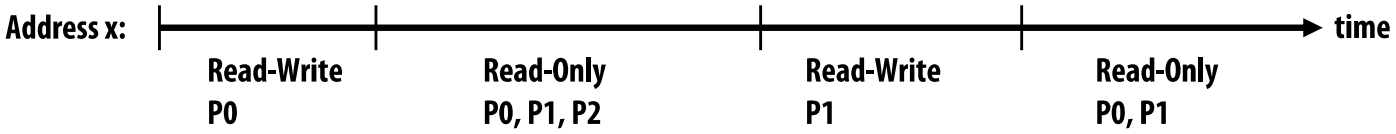
Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read x	S	--	--	BusRd	Memory
2. P3 read x	S	--	S	BusRd	Memory
3. P3 write x	I	--	M	BusRdX	Memory
4. P1 read x	S	--	S	BusRd	P3's cache
5. P2 read x	S	S	S	BusRd	Memory
6. P2 write x	I	M	I	BusRdX	Memory

- Single writer, multiple reader protocol
- Why do you need Modified to Shared?
- Communication increases memory latency

How Does MSI Satisfy Cache Coherence?

1. **Single-Writer, Multiple-Read (SWMR) Invariant**

2. **Data-Value Invariant (write serialization)**



Summary: MSI

- A line in the M state can be modified without notifying other caches
  - No other caches have the line resident, so other processors cannot read these values
  - (without generating a memory read transaction)

- Processor can only write to lines in the M state
  - If processor performs a write to a line that is not exclusive in cache, cache controller must first broadcast a read-exclusive transaction to move the line into that state
  - Read-exclusive tells other caches about impending write

  ("you can't read any more, because I'm going to write")
  - Read-exclusive transaction is required even if line is valid (but not exclusive… it's in the S state) in processor's local cache (why?)
  - Dirty state implies exclusive

- When cache controller snoops a "read exclusive" for a line it contains
  - Must invalidate the line in its cache
  - Because if it didn't, then multiple caches will have the line

  (and so it wouldn't be exclusive in the other cache!)

MESI invalidation protocol

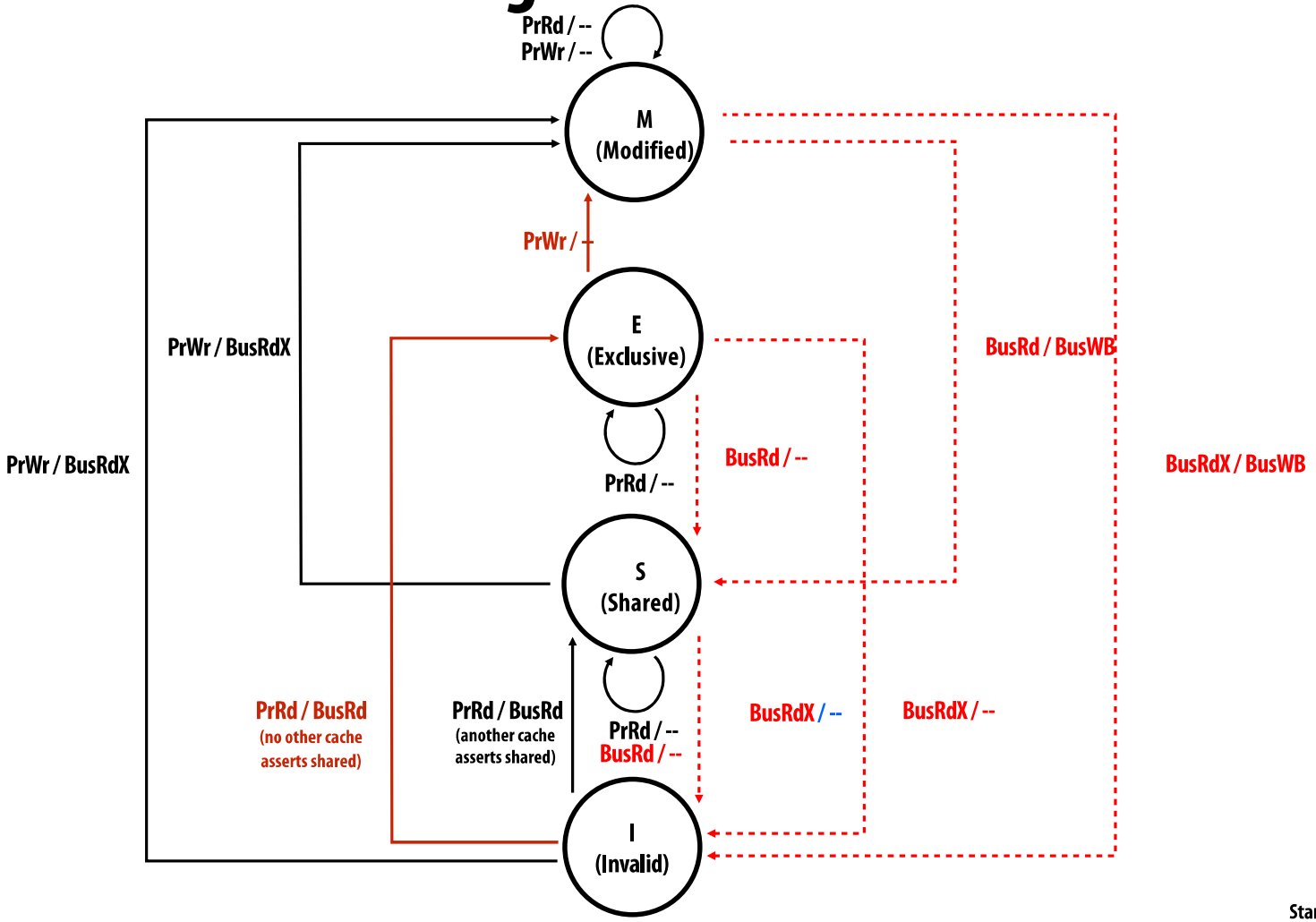
- MSI requires two interconnect transactions for the common case of reading an address, then writing to it
  - Transaction 1: BusRd to move from I to S state
  - Transaction 2: BusRdX to move from S to M state

MESI, not Messi!

- This inefficiency exists even if application has no sharing at all

- Solution: add additional state E ("exclusive clean")
  - Line has not been modified, but only this cache has a copy of the line
  - Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)
  - Upgrade from E to M does not require an bus transaction

MESI state transition diagram

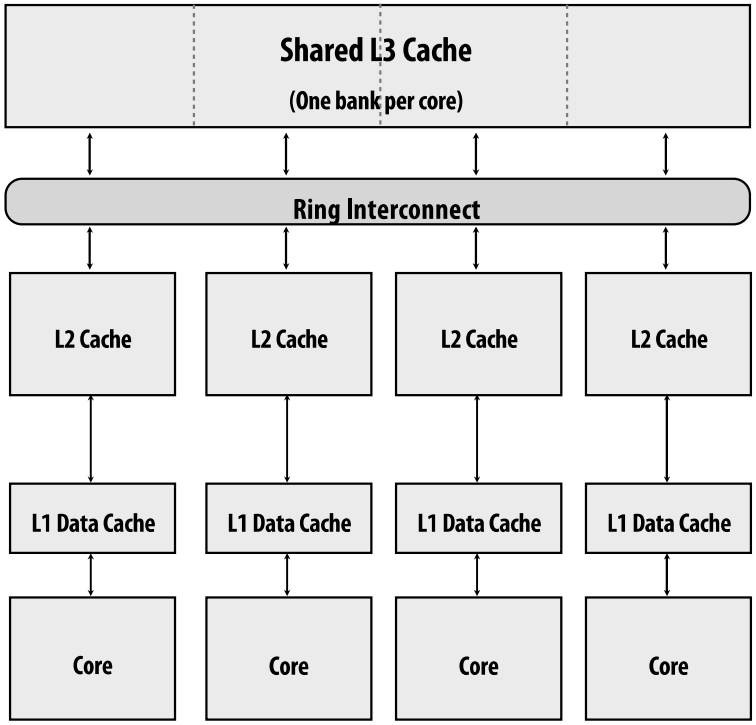


Scalable cache coherence using directories

- Snooping schemes broadcast coherence messages to determine the state of a line in the other caches: not scalable
- Alternative idea: avoid broadcast by storing information about the status of the line in one place: a "directory"
  - The directory entry for a cache line contains information about the state of the cache line in all caches.
  - Caches look up information from the directory as necessary
  - Cache coherence is maintained by point-to-point messages between the caches on a "need to know" basis (not by broadcast mechanisms)

- Still need to maintain invariants
  - SWMR
  - Write serialization

Directory coherence in Intel Core i7 CPU



- L3 serves as centralized directory for all lines in the L3 cache
  - Serialization piont

(Since L3 is an inclusive cache, any line in L2 is guaranteed to also be resident in L3)

- Directory maintains list of L2 caches containing line
- Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line

(Core i7 interconnect is a ring, it is not a bus)

- Directory dimensions:
  - P=4
  - M = number of L3 cache lines

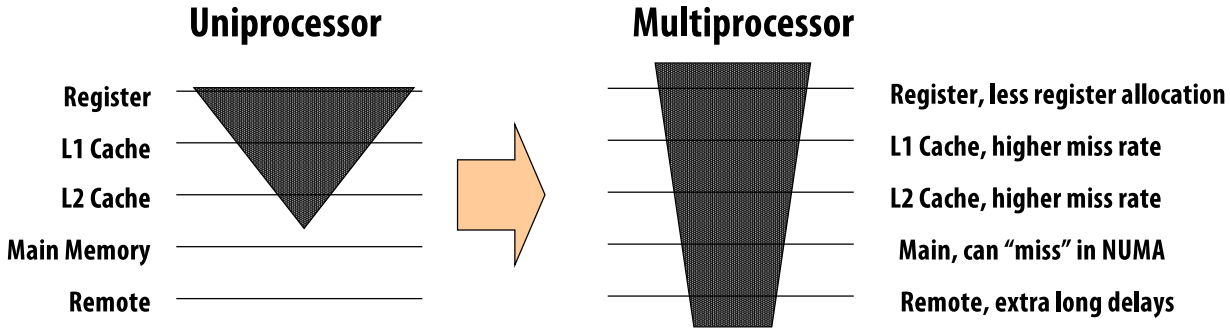
Implications of cache coherence to the programmer

Communication Overhead

- Communication time is a key parallel overhead
  - Appears as increased memory access time in multiprocessor
    - Extra main memory accesses in UMA systems
      - Must determine increase in cache miss rate vs. uniprocessor
    - Some accesses have higher latency in NUMA systems
      - Only a fraction of a % of these can be significant!

$$AMAT_{Multiprocessor} > AMAT_{Uniprocessor}$$

Average Memory Access Time (AMAT) = $\sum_0^n$ frequency of access $\times$ latency of access



Core i7 Xeon 5500 Series Data Source Latency (approx.)
L1 hit, ~4 cycles
L2 hit, ~10 cycles
L3 hit, line unshared ~40 cycles
L3 hit, shared line in another core ~65 cycles
L3 hit, modified in another core ~75 cycles remote
Local DRAM ~30 ns (~120 cycles)
Remote DRAM ~100 ns (~400 cycles)

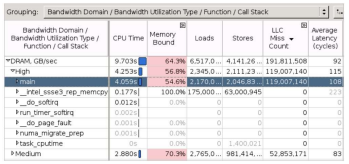
Use VTune to learn about memory system performance

Memory Access Analysis for Cache Misses and High Bandwidth Issues

Use the Intel® VTune™ Profiler's Memory Access analysis to identify memory-related issues, like NUMA problems and bandwidth-limited accesses, and attribute performance events to memory objects (data structures), which is provided due to instrumentation of memory allocations/de-allocations and getting static/global variables from symbol information.

NOTE:
Intel® VTune™ Profiler is a new renamed version of the Intel® VTune™ Amplifier.

How It Works



Grouping: Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack

Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
▾DRAM, GB/sec	9.703s	64.3%	6,517,0…	4,141,26…	191,811,508	92
▾High	4.253s	56.8%	2,345,0…	2,111,23…	119,007,140	115
main	4.058s	54.6%	2,170,0…	2,046,83…	119,007,140	108
▸__intel_ssse3_rep_memcpy	0.177s	100.0%	175,000…	63,000,945	0	223
▸__do_softirq	0.012s	0.0%	0	0	0	0
▸run_timer_softirq	0.002s		0	0	0	0
▸__do_page_fault	0.001s	0.0%	0	0	0	0
▸numa_migrate_prep	0.001s	0.0%	0	0	0	0
▸task_cputime	0s	0.0%	0	1,400,021	0	3
▸Medium	2.880s	70.3%	2,765,0…	981,414,…	52,853,171	83

Memory Access analysis type uses hardware event-based sampling to collect data for the following metrics:

- **Loads** and **Stores** metrics that show the total number of loads and stores
- **LLC Miss Count** metric that shows the total number of last-level cache misses
  - **Local DRAM Access Count** metric that shows the total number of LLC misses serviced by the local memory
  - **Remote DRAM Access Count** metric that shows the number of accesses to the remote socket memory
  - **Remote Cache Access Count** metric that shows the number of accesses to the remote socket cache
- **Memory Bound** metric that shows a fraction of cycles spent waiting due to demand load or store instructions
  - **L1 Bound** metric that shows how often the machine was stalled without missing the L1 data cache
  - **L2 Bound** metric that shows how often the machine was stalled on L2 cache
  - **L3 Bound** metric that shows how often the CPU was stalled on L3 cache, or contended with a sibling core
  - **L3 Latency** metric that shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited)
  - **NUMA: % of Remote Accesses** metric shows percentage of memory requests to remote DRAM. The lower its value is, the better.
  - **DRAM Bound** metric that shows how often the CPU was stalled on the main memory (DRAM). This metric enables you to identify **DRAM Bandwidth Bound**, **UPI Utilization Bound** issues, as well as **Memory Latency** issues with the following metrics:
    - **Remote / Local DRAM Ratio** metric that is defined by the ratio of remote DRAM loads to local DRAM loads
    - **Local DRAM** metric that shows how often the CPU was stalled on loads from the local memory
    - **Remote DRAM** metric that shows how often the CPU was stalled on loads from the remote memory
    - **Remote Cache** metric that shows how often the CPU was stalled on loads from the remote cache in other sockets
- **Average Latency** metric that shows an average load latency in cycles

Unintended communication via false sharing

What is the potential performance problem with this code?

```
// allocate per-thread variable for local per-thread accumulation
int myPerThreadCounter[NUM_THREADS];
```

Why might this code be more performant?

```
// allocate per thread variable for local accumulation
struct PerThreadState {
  int myPerThreadCounter;
  char padding[CACHE_LINE_SIZE - sizeof(int)];
};
PerThreadState myPerThreadCounter[NUM_THREADS];
```

Demo: false sharing

```
void* worker(void* arg) {

    volatile int* counter = (int*)arg;

    for (int i=0; i<MANY_ITERATIONS; i++)
        (*counter)++;

    return NULL;
}
```

threads update a per-thread counter many times

```
void test1(int num_threads) {

    pthread_t threads[MAX_THREADS];
    int       counter[MAX_THREADS];

    for (int i=0; i<num_threads; i++)
         pthread_create(&threads[i], NULL,
                        &worker, &counter[i]);

     for (int i=0; i<num_threads; i++)
         pthread_join(threads[i], NULL);
}
```

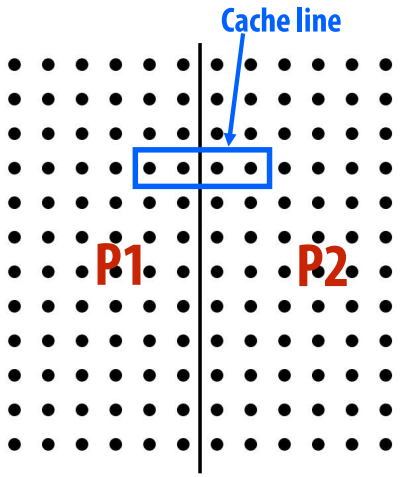
Execution time with num_threads=8 on 4-core system: 14.2 sec

```
struct padded_t {
    int  counter;
    char padding[CACHE_LINE_SIZE - sizeof(int)];
};

void test2(int num_threads) {

    pthread_t threads[MAX_THREADS];
    padded_t  counter[MAX_THREADS];


    for (int i=0; i<num_threads; i++)
        pthread_create(&threads[i], NULL,
                       &worker, &(counter[i].counter));

    for (int i=0; i<num_threads; i++)
        pthread_join(threads[i], NULL);
}
```

Execution time with num_threads=8 on 4-core system: 4.7 sec

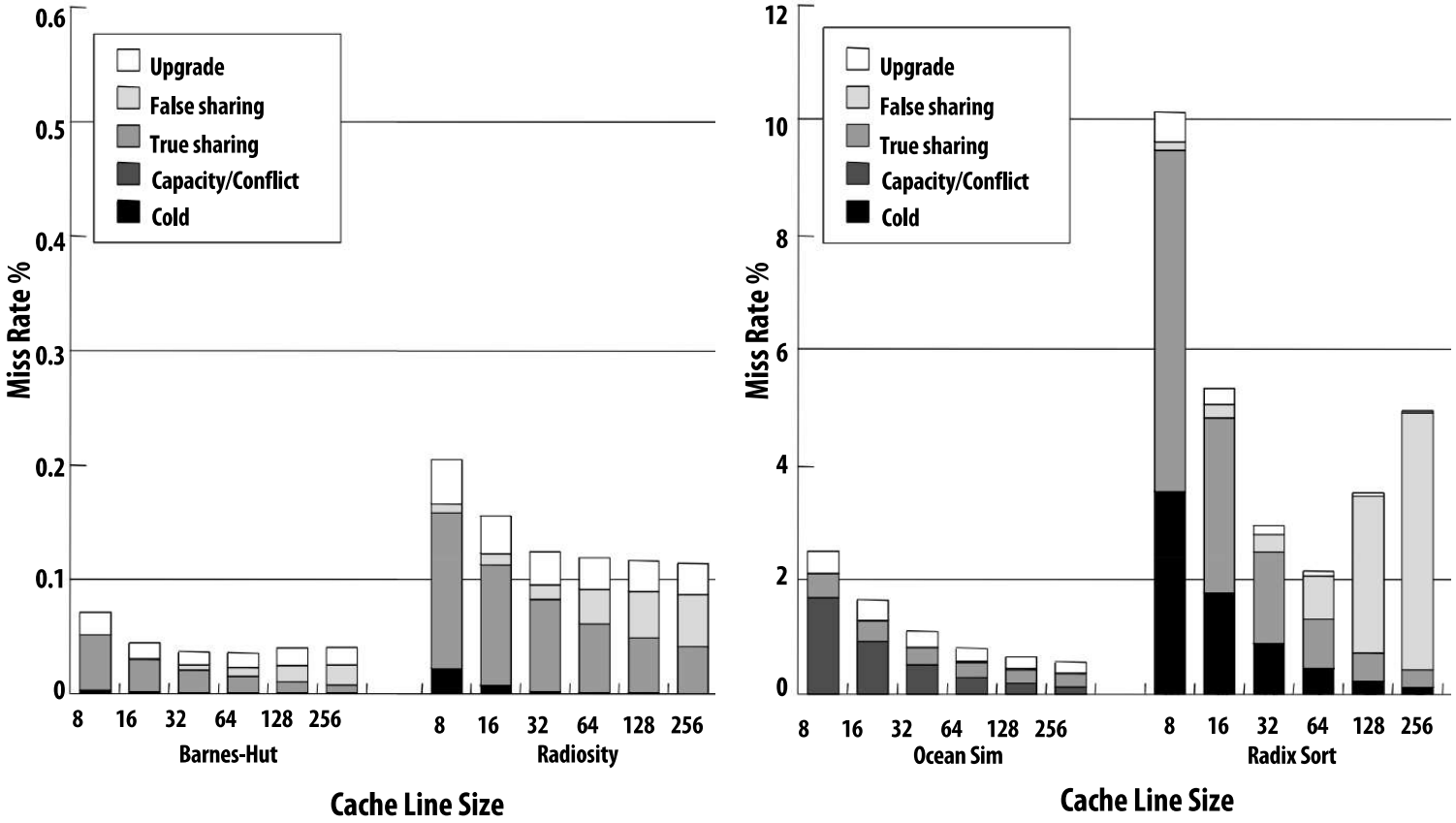
False sharing

- Condition where two processors write to different addresses, but addresses map to the same cache line
- Cache line "ping-pongs" between caches of writing processors, generating significant amounts of communication due to the coherence protocol
- No inherent communication, this is entirely artifactual communication (cachelines > 4B)
- False sharing can be a factor in when programming for cache-coherent architectures



Impact of cache line size on miss rate

Results from simulation of a 1 MB cache (four example applications)



* Note: I separated the results into two graphs because of different Y-axis scales

Figure credit: Culler, Singh, and Gupta

Summary: Cache coherence

- The cache coherence problem exists because the abstraction of a single shared address space is not implemented by a single storage unit
  - Storage is distributed among main memory and local processor caches
  - Data is replicated in local caches for performance

- Main idea of snooping-based cache coherence: whenever a cache operation occurs that could affect coherence, the cache controller broadcasts a notification to all other cache controllers in the system
  - Challenge for HW architects: minimizing overhead of coherence implementation
  - Challenge for SW developers: be wary of artifactual communication due to coherence protocol (e.g., false sharing)

- Scalability of snooping implementations is limited by ability to broadcast coherence messages to all caches!
  - Scaling cache coherence via directory-based approaches