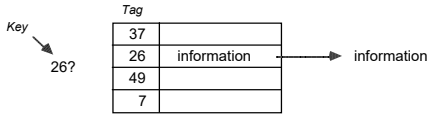
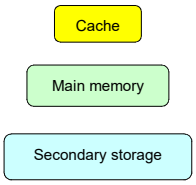
Cache memories

[§5.1] A *cache* is a small, fast memory which is *transparent* to the processor.

- The cache duplicates information that is in main memory.
- With each data block in the cache, there is associated an *identifier* or *tag*. This allows the cache to be *content addressable*.



- Caches are smaller and faster than main memory.
- *Secondary storage*, on the other hand, is larger and slower.



- A *cache miss* is the term analogous to a page fault. It occurs when a referenced word is not in the cache.
  - Cache misses must be handled much more quickly than page faults. Thus, they are handled in hardware.
- Caches can be *organized* according to four different strategies:
  - Direct
  - Fully associative
  - Set associative
  - Sectored

- A cache implements several different *policies* for retrieving and storing information, one in each of the following categories:
  - *Placement policy*—determines where a block is placed when it is brought into the cache.
  - *Replacement policy*—determines what information is purged when space is needed for a new entry.
  - *Write policy*—determines how soon information in the cache is written to lower levels in the memory hierarchy.

Cache memory organization

[§5.2] Information is moved into and out of the cache in *blocks*. When a block is in the cache, it occupies a cache *line*. Blocks are usually larger than one byte,

- to take advantage of locality in programs, and
- because memory may be organized so that it can overlap transfers of several bytes at a time.

The block size is the same as the line size of the cache.

A *placement policy* determines where a particular block can be placed when it goes into the cache. E.g., is a block of memory eligible to be placed in any line in the cache, or is it restricted to a single line?

In our examples, we assume—

- The cache contains 2048 bytes,
  with 16 bytes per line
  Thus it has 128 lines.
- Main memory is made up of 256K bytes, or 16384 blocks.
  Thus an address consists of 18 bits

We want to structure the cache to achieve a high *hit ratio.*

- *Hit*—the referenced information is in the cache.
- *Miss*—referenced information is not in cache, must be read in from main memory.

$$\text{Hit ratio} \equiv \frac{\text{Number of hits}}{\text{Total number of references}}$$

We will study caches that have three different placement policies (direct, fully associative, set associative).

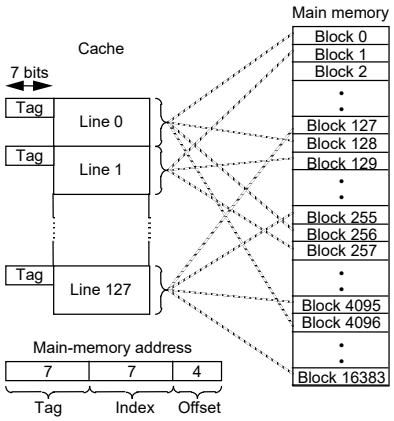
Direct

Only 1 choice of where to place a block.

block $i \rightarrow$ line $i \bmod 128$

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order seven bits of the main-memory address of the block.



00000000000011010

To search for a word in the cache,

1. Determine what line to look in (easy; just select bits 10–4 of the address).
2. Compare the leading seven bits (bits 17–11) of the address with the tag of the line. If it matches, the block is in the cache.
3. Select the desired bytes from the line.

Advantages:

Fast lookup (only one comparison needed).

Cheap hardware (only one tag needs to be checked).

Easy to decide where to place a block

Disadvantage: Contention for cache lines.

Exercise: What would the size of the tag, index, and offset fields be if—

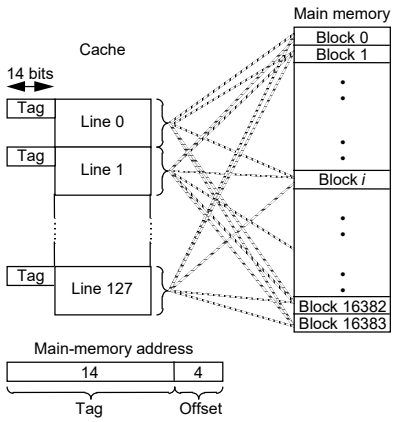
- the line size from our example were doubled, without changing the size of the cache?
- the cache size from our example were doubled, without changing the size of the line?
- an address were 32 bits long, but the cache size and line size were the same as in the example?

Fully associative

Any block can be placed in *any* line in the cache.

This means that we have 128 choices of where to place a block.

block $i \rightarrow$ any free (or purgeable) cache location



Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order *fourteen* bits of the main-memory address of the block.

To search for a word in the cache,

1. Simultaneously compare the leading 14 bits (bits 17–4) of the address with the tag of all lines. If it matches any one, the block is in the cache.
2. Select the desired bytes from the line.

Advantages:

Minimal contention for lines.

Wide variety of replacement algorithms feasible.

Exercise: What would the size of the tag and offset fields be if—

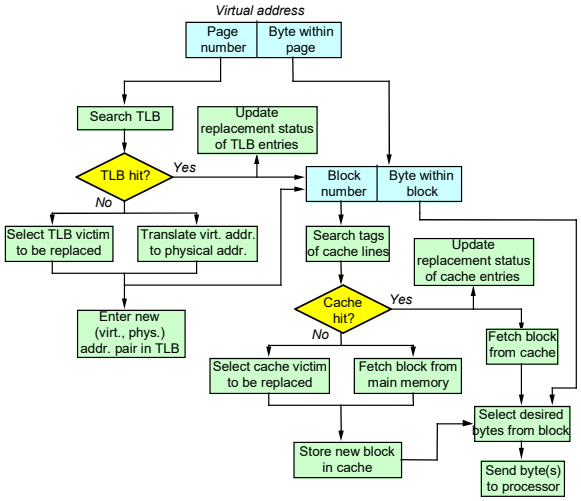
- the line size from our example were doubled, without changing the size of the cache?

- the cache size from our example were doubled, without changing the size of the line?
- an address were 32 bits long, but the cache size and line size were the same as in the example?

Disadvantage:

The most expensive of all organizations, due to the high cost of associative-comparison hardware.

A flowchart of cache operation: The process of searching a fully associative cache is very similar to using a directly mapped cache. Let us consider them in detail.



Which steps would be different if the cache were directly mapped?
Search tag of cache *line*.
Don't need to update replacement status.

Set associative

$1 < n < 128$ choices of where to place a block.

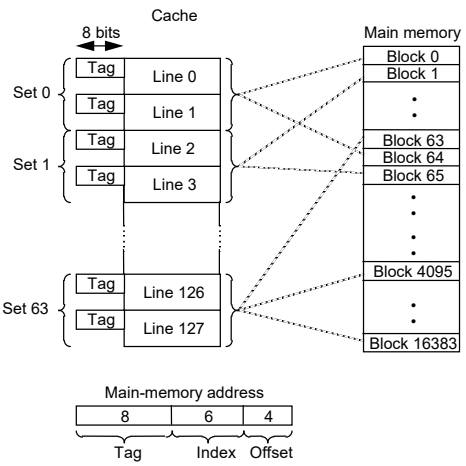
A compromise between direct and fully associative strategies.

The cache is divided into s sets, where s is a power of 2.

$$\text{block } i \rightarrow \text{any line in set } i \bmod s$$

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order *eight* bits of the main-memory address of the block. (The next six bits can be derived from the set number.)



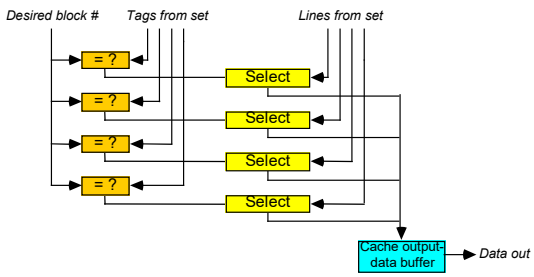
Exercise: What would the size of the tag, index, and offset fields be if—

- the line size from our example were doubled, without changing the size of the cache?
- the set size from our example were doubled, without changing the size of a line or the cache?
- the cache size from our example were doubled, without changing the size of the line or a set?
- an address were 32 bits long, but the cache size and line size was the same as in the example?

To search for a word in the cache,

1. Select the proper set (i mod s).
2. Simultaneously compare the leading 8 bits (bits 17–10) of the address with the tag of all lines in the set. If it matches any one, the block is in the cache.

   At the same time, the (first bytes of) the lines are also being read out so they will be accessible at the end of the cycle.
3. If a match is found, gate the data from the proper block to the cache-output buffer.
4. Select the desired bytes from the line



- All reads from the cache occur as early as possible, to allow maximum time for the comparison to take place.
- Which line to use is decided late, after the data have reached high-speed registers, so the processor can receive the data fast.

Factors influencing line lengths:

- Long lines ⇒ higher hit ratios.
- Long lines ⇒ less memory devoted to tags.
- Long lines ⇒ longer memory transactions (undesirable in a multiprocessor).
- Long lines ⇒ more write-backs (explained below).

For most machines, line sizes between 32 and 128 bytes perform best.

If there are *b* lines per set, the cache is said to be *b-way* set associative. How many way associative was the example above?

The logic to compare 2, 4, or 8 tags simultaneously can be made quite fast.

But as *b* increases beyond that, cycle time starts to climb, and the higher cycle time begins to offset the increased associativity.

Almost all L1 caches are less than 8-way set-associative. L2 caches often have higher associativity.

Two-level caches

Write policy

[§5.2.3] Answer these questions, based on the text.

What are the two write policies mentioned in the text?

Which one is typically used when a block is to be written to main memory, and why?

Which one can be used when a block is to be written to a lower level of the cache, and why?

Can you explain what error correction has to do with the choice of write policy?

Explain what a parity bit has to do with this.

Principle of inclusion

[§5.2.4] To analyze a second-level cache, we use the *principle of inclusion*—a large second-level cache includes everything in the first-level cache.

We can then do the analysis by assuming the first-level cache did not exist, and measuring the hit ratio of the second-level cache alone.

How should the line length in the second-level cache relate to the line length in the first-level cache?

When we measure a two-level cache system, two miss ratios are of interest:

- The *local miss rate* for a cache is the

  $$\frac{\text{\# misses experienced by the cache}}{\text{number of incoming references}}$$

  To compute this ratio for the L2 cache, we need to know the number of misses in the L1

- The *global miss rate* of the cache is

  $$\frac{\text{\# L2 misses}}{\text{\# of references made by processor}}$$

  This is the primary measure of the L2 cache.

What conditions need to be satisfied in order for inclusion to hold?

- L2 associativity must be ≥ L1 associativity, irrespective of the number of sets.

  Otherwise, more entries in a particular set could fit into the L1 cache than the L2 cache, which means the L2 cache couldn't hold everything in the L1 cache.
- The number of L2 sets has to be ≥ the number of L1 sets, irrespective of L2 associativity.

  (Assume that the L2 line size is ≥ L1 line size.)

  If this were not true, multiple L1 sets would depend on a single L2 set for backing store. So references to one L1 set could affect the backing store for another L1 set.
- All reference information from L1 is passed to L2 so that it can update its replacement bits.

Even if all of these conditions hold, we still won't have logical inclusion if L1 is write-back. (However, we will still have *statistical inclusion*—L2 *usually* contains L1 data.)

NC STATE UNIVERSITY

The Cache-Coherence Problem

Lecture 12
(Chapter 6)

1

Outline

- **Bus-based multiprocessors**
- The cache-coherence problem
- Peterson's algorithm
- Coherence vs. consistency

2

Shared vs. Distributed Memory

- What is the difference between …
  - SMP
  - NUMA
  - Cluster ?

3

Small to Large Multiprocessors

- **Small scale** (2–30 processors): shared memory
  - Often on-chip: shared memory (+ perhaps shared cache)
  - Most processors have MP support out of the box
  - Most of these systems are bus-based
  - Popular in commercial as well as HPC markets
- **Medium scale** (64–256): shared memory and clusters
  - Clusters are cheaper
  - Often, clusters of SMPs
- **Large scale** (> 256): few shared memory and many clusters
  - SGI Altix 3300: 512-processor shared memory (NUMA)
  - Large variety on custom/off-the-shelf components such as interconnection networks.
    - Beowulf clusters: fast Ethernet
    - Myrinet: fiber optics
    - IBM SP2: custom

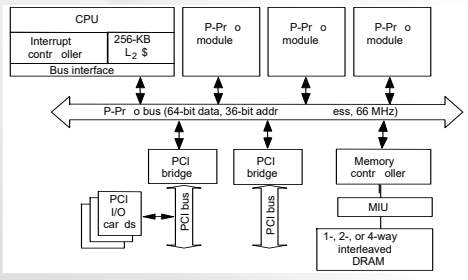
4

Shared Memory vs. No Shared Memory

- Advantages of shared-memory machines (vs. distributed memory w/same total memory size)
  - Support shared-memory programming
    - Clusters can also support it via *software shared virtual memory*, but with much coarser granularity and higher overheads
  - Allow fine-grained sharing
    - You can't do this with messages—there's too much overhead to share small items
  - Single OS image
- Disadvantage of shared-memory machines
  - Cost of providing shared-memory abstraction

5

A Bus-Based Multiprocessor



6

Outline

- Bus-based multiprocessors
- **The cache-coherence problem**
- Peterson's algorithm
- Coherence vs. consistency

7

Will This Parallel Code Work Correctly?

```
sum = 0;
begin parallel
for (i=1; i<=2; i++) {
    lock(id, myLock);
    sum = sum + a[i];
    unlock(id, myLock);
end parallel
print sum;

Suppose a[1] = 3 and
        a[2] = 7
```

Two issues:

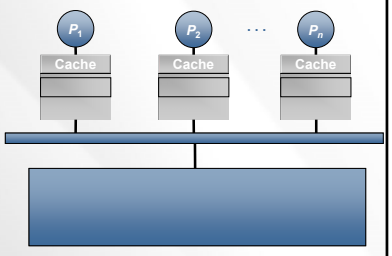
- **Will it print `sum = 10`?**
- **How can it support locking correctly?**

8

The Cache-Coherence Problem

```
sum = 0;
begin parallel
for (i=1; i<=2; i++) {
    lock(id, myLock);
    sum = sum + a[i];
    unlock(id, myLock);
end parallel
print sum;

Suppose a[1] = 3 and
a[2] = 7
```

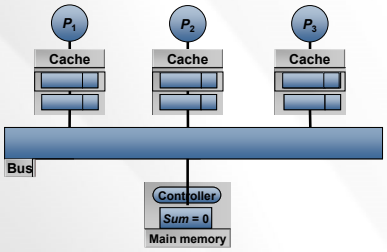


- **Will it print `sum = 10`?**

9

Cache-Coherence Problem Illustration

Start state. All caches empty and main memory has *Sum* = 0.



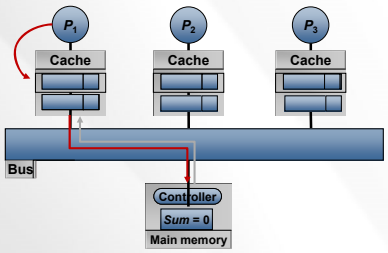
Trace

P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

10

Cache-Coherence Problem Illustration

P_1 reads *Sum* from memory.



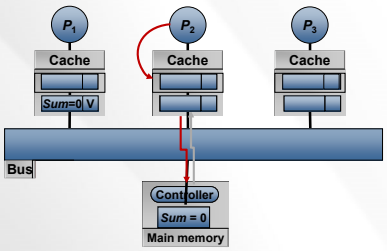
Trace

P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

11

Cache-Coherence Problem Illustration

P_2 reads. Let's assume this comes from memory too.

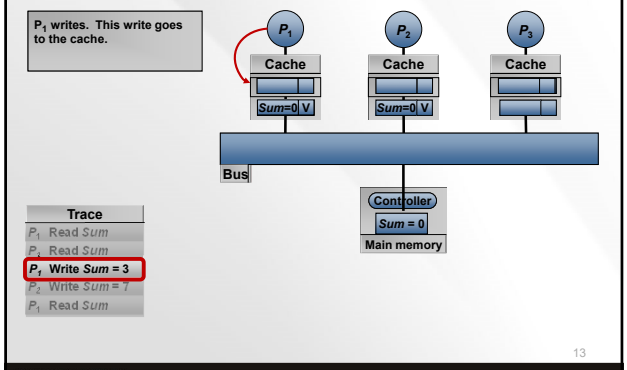


Trace

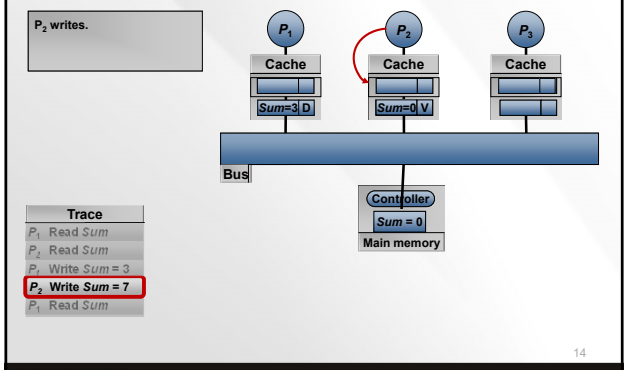
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

12

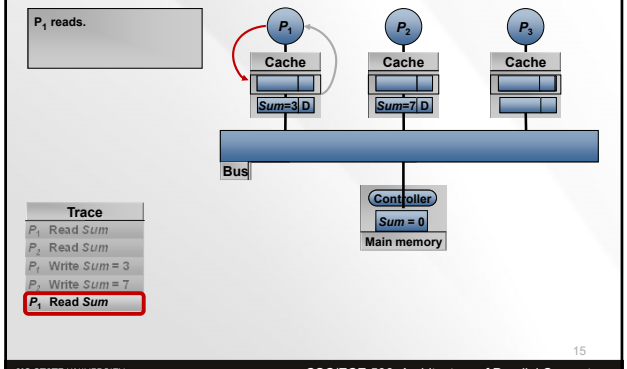
Cache-Coherence Problem Illustration



Cache-Coherence Problem Illustration



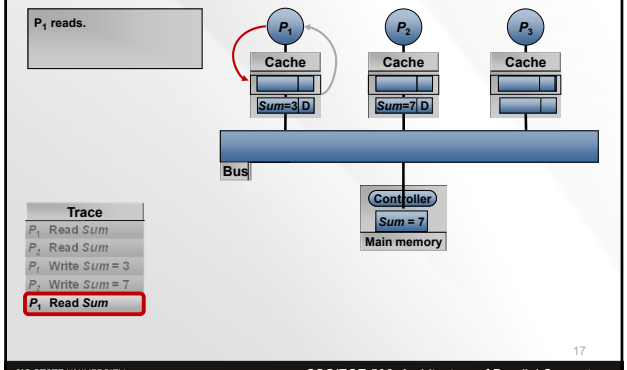
Cache-Coherence Problem Illustration



Cache-Coherence Problem

- Do P1 and P2 see the same sum?
- Does it matter if we use a WT cache?
- What if we do not have caches, or sum is uncacheable. Will it work?
- The code given at the start of the animation does not exhibit the same coherence problem shown in the animation. Explain why.

Write-Through Cache Does Not Work



Software Lock Using a Flag

- Here's simple code to implement a lock:

```
void lock (int process, int lvar) {      // process is 0 or 1
  while (lvar == 1) {} ;
  lvar = 1;
}

void unlock (int process, int lvar) {
  lvar = 0;
}
```

- Will this guarantee mutual exclusion?
- Let's look at an algorithm that will …

Outline

- Bus-based multiprocessors
- The cache-coherence problem
- **Peterson's algorithm**
- Coherence vs. consistency

Peterson's Algorithm

```
int turn;
int interested[n];  // initialized to false

void lock (int process, int lvar) {     // process is 0 or 1
  int other = 1 - process;
  interested[process] = TRUE;
  turn = other;
  while (turn == other && interested[other] == TRUE) {} ;
}
// Post: turn != other or interested[other] == FALSE

void unlock (int process, int lvar) {
  interested[process] = FALSE;
}
```

- Acquisition of **`lock()`** occurs only if
  1. **`interested[other] == FALSE`**: either the other process has not competed for the lock, or it has just called **`unlock()`**, or
  2. **`turn != other`**: the other process is competing, has set the turn to *our* process, and will be blocked in the **`while()`** loop

No Race

```
// Proc 0
interested[0] = TRUE;
turn = 1;
while (turn==1 && interested[1]==TRUE)
  {};
// since interested[1] starts out FALSE,
// Proc 0 enters critical section
```

```
// Proc 1
interested[1] = TRUE;
turn = 0;
while (turn==0 && interested[0]==TRUE)
  {};
// since turn==0 && interested[0]==TRUE
// Proc 1 waits in the loop until Proc 0
// releases the lock
```

```
// unlock
interested[0] = FALSE;
```

```
// now Proc 1 can exit the loop and
// acquire the lock
```


Race

```
// Proc 0
interested[0] = TRUE;
turn = 1;

while (turn==1 && interested[1]==TRUE)
  {};
// since turn == 0,
// Proc 0 enters critical section

// unlock
interested[0] = FALSE;
```

```
// Proc 1
interested[1] = TRUE;

turn = 0;

while (turn==0 && interested[0]==TRUE)
  {};
// since turn==0 && interested[0]==TRUE
// Proc 1 waits in the loop until Proc 0
// releases the lock

// now Proc 1 can exit the loop and
// acquire the lock
```


When Does Peterson's Alg. Work?

- Correctness depends on the global order of

```
A: interested[process] = TRUE;
B: turn = other;
```

- Thus, it will not work if—
  - The *compiler* reorders the operations
    - There's no data dependence, so unless the compiler is notified, it may well reorder the operations
    - This prevents compiler from using aggressive optimizations used in serial programs
  - The *architecture* reorders the operations
    - Write buffers, memory controller
    - Network delay for statement A
    - If **`turn`** and **`interested[]`** are cacheable, A may result in cache miss, but B in cache hit
- This is called the memory-consistency problem.

Race on a Non-Sequentially Consistent Machine

```
// Proc 0
interested[0] = TRUE;
```

```
// Proc 1

interested[1] = TRUE;
turn = 0;
```

```
turn = 1;
while (turn==1 && interested[1]==TRUE)
  {};
```

```
while (turn==0 && interested[0]==TRUE)
  {};
```

Race on a Non-Sequentially Consistent Machine

```
// Proc 0                              // Proc 1
interested[0] = TRUE;

                                       turn = 0;          reordered

turn = 1;
while (turn==1 && interested[1]==TRUE)
  ();
// since interested[1] == FALSE,       interested[1] = TRUE;
// Proc 0 enters critical section      while (turn==0 && interested[0]==TRUE)
                                         ();
                                       // since turn==1,
                                       // Proc 1 enters critical section
```

Can you explain what has gone wrong here?

Coherence vs. Consistency

Cache coherence	Memory consistency
Deals with the ordering of operations to a *single* memory location.	Deals with the ordering of operations to *different memory locations.*

Coherence vs. Consistency

Cache coherence	Memory consistency
Deals with the ordering of operations to a *single* memory location.	Deals with the ordering of operations to *different memory locations.*
Tackled by hardware • using coherence protocols. • Hw. alone guarantees correctness but with varying performance	Tackled by consistency models • supported by hardware, but • software must conform to the model.

Coherence vs. Consistency

Cache coherence	Memory consistency
Deals with the ordering of operations to a *single* memory location.	Deals with the ordering of operations to *different memory locations.*
Tackled by hardware • using coherence protocols. • Hw. alone guarantees correctness but with varying performance	Tackled by consistency models • supported by hardware, but • software must conform to the model.
All protocols realize same abstraction • A program written for 1 protocol can run w/o change on any other.	Models provide diff. abstractions • Compilers must be aware of the model (no reordering certain operations …). • Programs must "be careful" in using shared variables.

Two Approaches to Consistency

- Sequential consistency
  - Multi-threaded codes for uniprocessors automatically run correctly
  - How? Every shared R/W completes globally in program order
  - Most intuitive but worst performance
- Relaxed consistency models
  - Multi-threaded codes for uniprocessor need to be ported to run correctly
  - Additional instruction (memory fence) to ensure global order between 2 operations

Cache Coherence

- Do we need caches?
  - Yes, to reduce average data access time.
  - Yes, to reduce bandwidth needed for bus/interconnect.
- Sufficient conditions for coherence:
  - Notation: Request$_{proc}$(*data*)
  - ***Write propagation***:
    - Rd$_i$(*X*) must return the "latest" Wr$_j$(*X*)
  - ***Write serialization***:
    - Wr$_i$(*X*) and Wr$_j$(*X*) are seen in the same order by everybody
      - e.g., if I see w2 after w1, you shouldn't see w2 before w1
      - ➔ There must be a **global ordering** of memory operations *to a single location*
  - Is there a need for *read serialization*?

A Coherent Memory System: Intuition

- Uniprocessors
  - Coherence between I/O devices and processors
  - Infrequent, so software solutions work
    - uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches
- But coherence problem much more critical in multiprocessors
  - Pervasive
  - Performance-critical
  - Necessitates a hardware solution
- * Note that "latest write" is ambiguous.
  - Ultimately, what we care about is that any write is propagated everywhere in the same order.
  - Synchronization defines what "latest" means.

Summary

- Shared memory with caches raises the problem of cache coherence.
  - Writes to the same location must be seen in the same order everywhere.
- But this is not the only problem
  - Writes to *different* locations must also be kept in order if they are being depended upon for synchronizing tasks.
  - This is called the memory-consistency problem

NC STATE UNIVERSITY

Coherence and Consistency

Lecture 13
(Chapter 7)

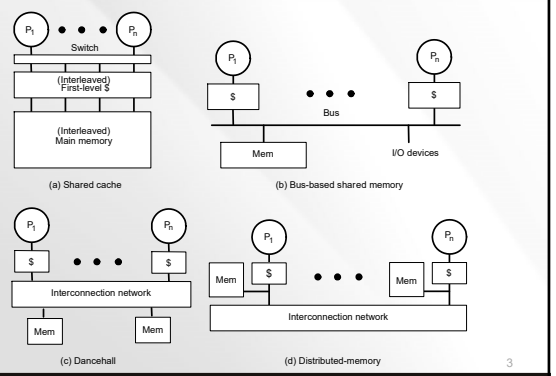
1

Outline

- **Bus-based coherence**
- Invalidation vs. update coherence protocols
- Memory consistency
  - Sequential consistency

2

Several Configurations for a Memory System



3

Assume a Bus-Based SMP

- Built on top of two fundamentals of uniprocessor system
  - Bus transactions
  - Cache-line finite-state machine
- Uniprocessor bus transaction:
  - Three phases: arbitration, command/address, data transfer
  - All devices observe addresses, one is responsible
- Uniprocessor cache states:
  - Every cache line has a finite-state machine
  - In WT+write no-allocate: Valid, Invalid states
  - WB: Valid, Invalid, Modified ("Dirty")
- Multiprocessors extend both these somewhat to implement coherence

4

Snoop-Based Coherence on a Bus

- *Basic Idea*
  - Assign a snooper to each processor so that all bus transactions are visible to all processors ("snooping").
  - Processors (via cache controllers) change line states on relevant events.

5

Snoop-Based Coherence on a Bus

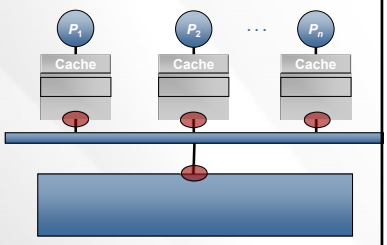
- *Basic Idea*
  - Assign a snooper to each processor so that all bus transactions are visible to all processors ("snooping").
  - Processors (via cache controllers) change line states on relevant events.
- *Implementing a Protocol*
  - Each **cache controller** reacts to processor and bus events:
    - Takes actions when necessary
      - Updates state, responds with data, generates new bus transactions
  - The **memory controller** also snoops bus transactions and returns data only when needed
  - Granularity of coherence is typically one cache line/block
    - Same granularity as in transfer to/from cache

6

Coherence with Write-Through Caches

```
sum = 0;
begin parallel
for (i=0; i<2; i++) {
  lock(id, myLock);
  sum = sum + a[i];
  unlock(id, myLock);
end parallel
Print sum;

Suppose a[0] = 3 and a[1] = 7
```

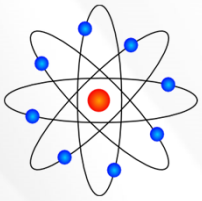


= Snooper

- What happens when we snoop a write?
  - Write-update protocol: write is immediately propagated ***or***
  - Write-invalidation protocol: causes miss on later access, and memory up-to-date via write-through

Snooper Assumptions

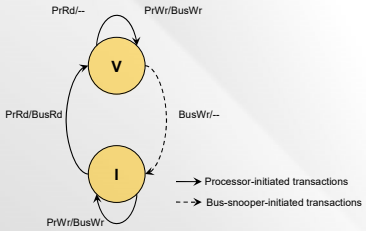
- Atomic bus
- Writes occur in program order



Transactions

- To show what's going on, we will use diagrams involving—
  - Processor transactions
    - PrRd
    - PrWr
  - Snooped bus transactions
    - BusRd
    - BusWr

Write-Through State-Transition Diagram



write-through
no-write-allocate
write invalidate

How does this protocol guarantee write propagation?

How does it guarantee write serialization?

- Key: A write invalidates all other caches
- Therefore, we have:
  - Modified line: exists as V in only 1 cache
  - Clean line: exists as V in at least 1 cache
  - Invalid state represents invalidated line or not present in the cache

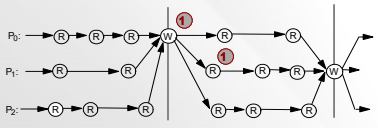
Is It Coherent?

- Write propagation:
  - through invalidation
  - then a cache miss, loading a new value
- Write serialization: Assume—
  - atomic bus
  - invalidation happens instantaneously
  - writes serialized by order in which they appear on bus (*bus order*)
    - So are invalidations
- Do reads see the latest writes?
  - Read misses generate bus transactions, so will get the last write
  - Read hits: do not appear on bus, but are preceded by
    - most recent write by this processor (self), or
    - most recent read miss by this processor
  - Thus, reads hits see latest written values (according to bus order)

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.

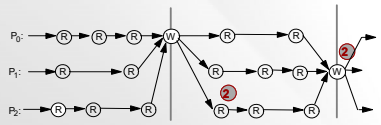


- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.
2. Write follows read or write M if M generates bus transaction and the transaction for the write follows that for M.

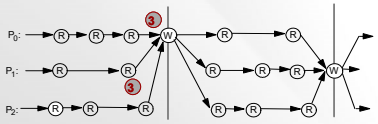


- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.
2. Write follows read or write M if M generates bus transaction and the transaction for the write follows that for M.
3. Write follows read if read does not generate a bus transaction and is not already separated from the write by another bus transaction.



- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Problem with Write-Through

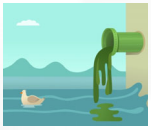
- Write-through can guarantee coherence, but it requires a lot of bandwidth.
  - Every write goes to the shared bus and memory
  - Example:
    200MHz, 1-CPI processor, and 15% instrs. are 8-byte stores
    Each processor generates 30M stores, or 240MB data, per second
    How many processors could a 1GB/s bus support without saturating?
  - Thus, unpopular for SMPs
- Write-back caches
  - Write hits do not go to the bus ⇒ reduce most write bus transactions
  - But now how do we ensure write propagation and serialization?

Lecture 14 Outline

- Bus-based coherence
- **Invalidation vs. update coherence protocols**
- Memory consistency
  - Sequential consistency

Dealing with "Dirty" Lines

- What does it mean to say a cache line is "dirty"?
  - That at least one of its words has been changed since it was brought in from main memory.
- Dirty in a uniprocessor vs. a multiprocessor
  - Uniprocessor:
    - Only need to keep track of *whether* a line has been modified.
  - Multiprocessor:
    - Keep track of *whether* line is modified.
    - Keep track of which cache *owns* the line.
  - Thus, a cache line must know whether it is—
    - **Exclusive**: "I'm the only one that has it, other than possibly main memory."
    - The **Owner**: "I'm responsible for supplying the block upon a request for it."



Invalidation vs. Update Protocols

- Question: What happens to a line if *another* processor changes one of its words?
  - It can be *invalidated.*
  - It can be *updated.*

Invalidation-Based Protocols

- Idea: When I write the block, invalidate everybody else ⇒ I get exclusive state.
- "Exclusive" means …
  - Can modify without notifying anyone else (i.e., without a bus transaction)
- But, before writing to it,
  - Must first get block in exclusive state
  - Even if block is already in state V, a bus transaction (Read Exclusive = RdX) is needed to invalidate others.
- What happens when a block is ejected from the cache?
  - if the block is not dirty?
  - if the block *is* dirty?

UPDATE! -Based Protocols

- Idea: If this block is written, send the new word to all other caches.
  - New bus transaction: Update
- Compared to invalidate, what are advs. and disads.?
- Advantages
  - Other processors don't miss on next access
  - Saves refetch: In invalidation protocols, they would miss & bus transaction.
  - Saves bandwidth: A single bus transaction updates several caches
- Disadvantages
  - Multiple writes by same processor cause multiple update transactions
    - In invalidation, first write gets exclusive ownership, other writes local

Invalidate versus Update

- Is a block written by one processor read by other processors before it is rewritten?
- Invalidation:
  - Yes → Readers will take a miss.
  - No → Multiple writes can occur without additional traffic.
    - Copies that won't be used again get cleared out.
- Update:
  - Yes → Readers will not miss if they had a copy previously
    - A single bus transaction will update all copies
  - No → Multiple useless updates, even to dead copies
- Invalidation protocols are much more popular.
  - Some systems provide both, or even hybrid

Lecture 14 Outline

- Bus-based coherence
- Invalidation vs. update coherence protocols
- **Memory consistency**
  - Sequential consistency

Let's Switch Gears to Memory Consistency

Coherence: Writes to a single location are visible to all in the same order

Consistency: Writes to multiple locations are visible to all in the same order

- Recall Peterson's algorithm (`turn= …; interested[process]=…`)
- When "multiple" means "all", we have sequential consistency (SC)

P_1	P_2
/*Assume initial values of A and flag are 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Sequential consistency (SC) corresponds to our intuition.
- Other memory consistency models do not obey our intuition!
- Coherence doesn't help; it pertains only to a single location

Another Example of Ordering

P_1	P_2
/*Assume initial values of `A` and `B` are `0` */	
(1a) `A = 1;`	(2a) `print B;`
(1b) `B = 2;`	(2b) `print A;`

- What do you think should be printed? You may think:
  - 1a, 1b, 2a, 2b ⇒ `{A=1, B=2}`
  - 1a, 2a, 2b, 1b ⇒ `{A=1, B=0}`
  - 2a, 2b, 1a, 1b ⇒ `{A=0, B=0}`

  } programmers' intuition: sequential consistency
  - Is `{A=0, B=2}` possible? Yes, suppose P2 sees: 1b, 2a, 2b, 1a e.g. evil compiler, evil interconnection.
- Whatever our intuition is, we need
  - an **ordering model** for clear semantics across different locations
  - as well as **cache coherence**!

  so programmers can reason about what results are possible.

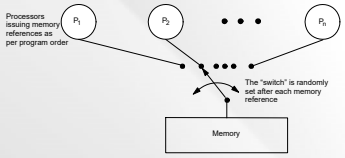
A Memory-Consistency Model …

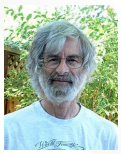
- Is a contract between programmer and system
  - Necessary to reason about correctness of shared-memory programs
- Specifies constraints on the order in which memory operations (from any process) can *appear to execute* with respect to one another
  - Given a load, constrains the possible values returned by it
- Implications for programmers
  - Restricts algorithms that can be used
  - e.g., Peterson's algorithm, home-brew synchronization will be incorrect in machines that do not guarantee SC
- Implications for compiler writers and computer architects
  - Determines how much accesses can be reordered.

Lecture 14 Outline

- Bus-based coherence
- Memory consistency
  - **Sequential consistency**
- Invalidation vs. update coherence protocols

Sequential Consistency





"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

- (as if there were no caches, and a single memory)
- Total order achieved by *interleaving* accesses from different processes
- Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others

What Really Is Program Order?

- Intuitively, the order in which operations appear in source code
- *Thus, we assume order as seen by programmer,*
  - *the compiler is prohibited from reordering memory accesses to shared variables.*
- Note that this is one reason parallel programs are less efficient than serial programs.

What Reordering Is Safe in SC?

What matters is the order in which code *appears to execute*, not the order in which it actually *executes*.

P_1	P_2
/*Assume initial values of `A` and `B` are `0` */	
(1a) `A = 1;`	(2a) `print B;`
(1b) `B = 2;`	(2b) `print A;`

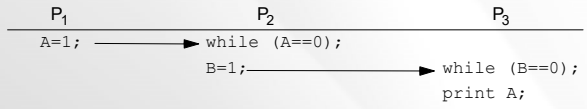
- Possible outcomes for `(A,B)`: `(0,0)`, `(1,0)`, `(1,2)`; impossible under SC: `(0,2)`
- *Proof:* By program order we know 1a → 1b and 2a → 2b
  - `A = 0` implies 2b → 1a, which implies 2a → 1b
  - `B = 2` implies 1b → 2a, which leads to a contradiction
- BUT, actual execution 1b →1a → 2b → 2a is SC, despite not being in program order
  - It produces the same result as 1a → 1b → 2a → 2b.
  - Actual execution 1b → 2a → 2b → 1a is not SC, as shown above
  - Thus, some reordering is possible, but difficult to reason that it ensures SC

Conditions for SC

- Two kinds of requirements
  - **Program order**
    - Memory operations issued by a process must appear to become visible (to others and itself) in program order.
  - **Global order**
    - Atomicity: One memory operation should appear to complete with respect to all processes before the next one is issued.
    - Global order: The same order of operations is seen by all processes.
- Tricky part: how to make writes atomic?
  - → Necessary to detect write completion
  - Read completion is easy: a read completes when the data returns
- Who should enforce SC?
  - Compiler should not change program order
  - Hardware should ensure program order and atomicity

Write Atomicity

- *Write Atomicity* ensures same write ordering is seen by all procs.
  - In effect, extends write serialization to writes from multiple processes



- Under SC, transitivity implies that `A` should print as 1. Without SC, why might it not?

Is the Write-Through Example SC?

- Assume no write buffers, or load-store bypassing
- Yes, it is SC, because of the atomic bus:
  - Any write and read misses (to *all locations)* are serialized by the bus into bus order.
  - If a read obtains value of write W, W is guaranteed to have completed since it caused a bus transaction
  - When write W is performed *with respect to any processor*, all previous writes in bus order have completed

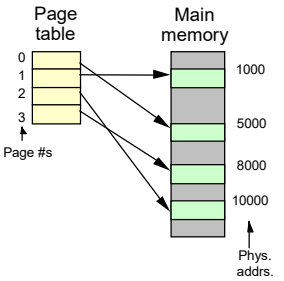
Summary

- One solution for small-scale multiprocessors is a shared bus.
- State-transition diagrams can be used to show how a cache-coherence *protocol* operates.
  - The simplest protocol is write-through, but it has performance problems.
- Sequential consistency guarantees that memory operations are seen in order throughout the system.
  - It is fairly easy to show whether a result is or is not sequentially consistent.
- The two main types of coherence protocols are invalidate and update.
  - Invalidate usually works better, because it frees up cache lines more quickly.

[§5.2.6] Translation Lookaside Buffers

The CPU generates *virtual* addresses, which correspond to locations in virtual memory.

In principle, the virtual addresses are translated to physical addresses using a page table.

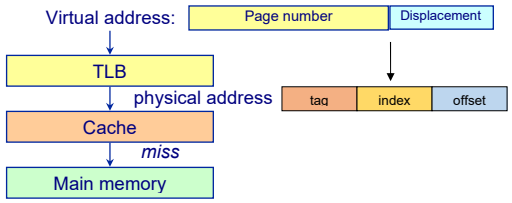


But this is too slow, so in practice, a *translation lookaside buffer* (TLB) is used.

It is like a special cache that is indexed by page number.

If there is a hit on a page number, then the address of the page in memory (called the *page-frame address*) is immediately obtained.

Therefore, the TLB and the cache must be accessed sequentially.

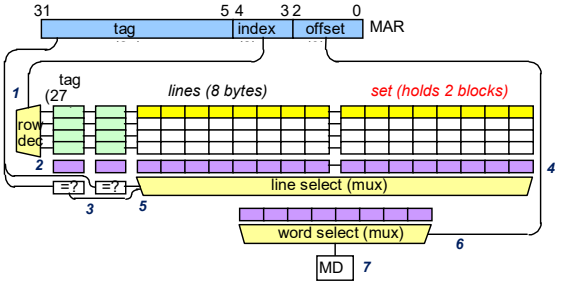


This adds an extra cycle in case of a hit.

(The page *displacement* is sometimes called the "page offset." But we will call it the displacement to avoid confusion with the block offset," which we just call "offset.")

How can we avoid wasting this time?

Let's look at what happens when a memory address is accessed.



What are the steps in cache access?

1. Access the set that could contain the address
2. Pull down the tags into the sense amplifiers
3. Compare the tags with the tag of the referenced addr.
4. Read all lines of the set into the sense amplifiers
5. Select the line that contains the sought-after addr.
6. Select the sought-after bytes/words to return
7. Return the bytes/words to the processor

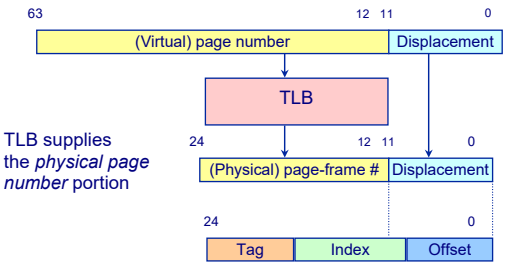
We always need to read lines into the sense amplifiers and then select the word (cf. the direct-mapped cache diagram in Lecture 4).

Now, if we know the index *before* address translation takes place, we can perform steps 1, 2, & 4 while address translation is occurring.

There is a tradeoff between speed and power efficiency.

- For power efficiency, which order should should steps 1 through 4 be performed in? 1, 2, 3, 4
- For maximum speed, which of steps 1 through 4 can be performed in parallel? 2 and 4

Let's take a look at address translation.



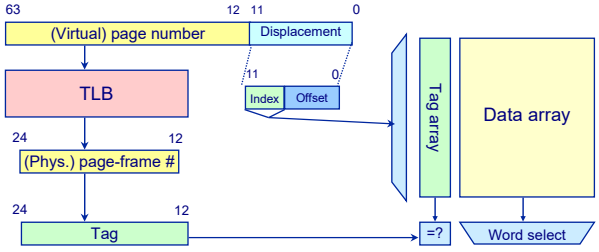
In this example, what is the page size (in bytes)? 2^{12}

How much physical memory is there? 2^{25}

Our goal is to allow the cache to be indexed before address translation completes.

In order to do that, we need to have the index field be *entirely contained* within the page displacement.

So, if the displacement is d bits wide, the width of the index is j bits, and the offset is k bits, we must have $j + k \le d$.



Cache hit time reduces from two cycles to one!

> … because the cache can now be *indexed* in parallel with TLB (although the tag match uses output from the TLB).

But there are some constraints...

- Suppose our cache is direct mapped. Then the index field just contains the line number. So, (line number || block offset) must fit inside the page displacement.

  What is the largest the cache can be? 1 page
- If we want to increase the size of the cache, what can we do? Increase associativity

Options:

- For new machines, select page size such that—

$$\text{page size} \ge \frac{\text{cache size}}{\text{associativity}}$$

- If page size is fixed, select associativity so that—

$$\text{associativity} \ge \frac{\text{cache size}}{\text{page size}}$$

Example: MC88110

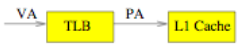
- Page size = 4KB
- I-cache, D-cache are both: 8KB, 2-way set-associative (4KB = 8KB / 2)

Example: VAX series

- Page size = 512B
- For a 16KB cache, need assoc. = (16KB / 512B) = 32-way set. assoc.!

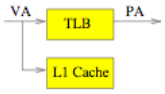
The textbook gives these three alternatives for cache indexing and tagging. Answer some questions about them.

Physically Indexed and Tagged



What's the main disadantage of physically indexed and tagged?

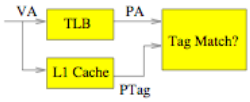
Virtually Indexed and Tagged



What is the organization we have just been discussing (in the last diagram)?

What is the main disadvantage of virtually indexed and tagged?

Virtually Indexed but Physically Tagged



Multilevel cache design

What are distinguishing features of the different cache levels of the four-level design (from 2013) illustrated on p. 135 of the textbook?

	Distinguish-ing feature	Size	Access time	Implement'n techology
L1 cache				
L2 cache				
L3 cache				
L4 cache				
Main mem.				

What are some advantages of a centralized cache?

Shorter wire length, since each processor only needs to be connected to a cache in one place

Interconnect to the L2 can be only one place.

What are some advantages of a banked structure?

The cache is not located in one place, so power (and heat) are distributed evenly around the chip.

A portion of the cache is closer to, and therefore, more quickly accessible to, each processor.

A single tile (core, L1 caches, 1 bank of L2) can be designed & stamped as many times as needed. That allows tiles to potentially be used over different generations of a chip.

Inclusion in multilevel caches

Answer these questions about inclusion policies.

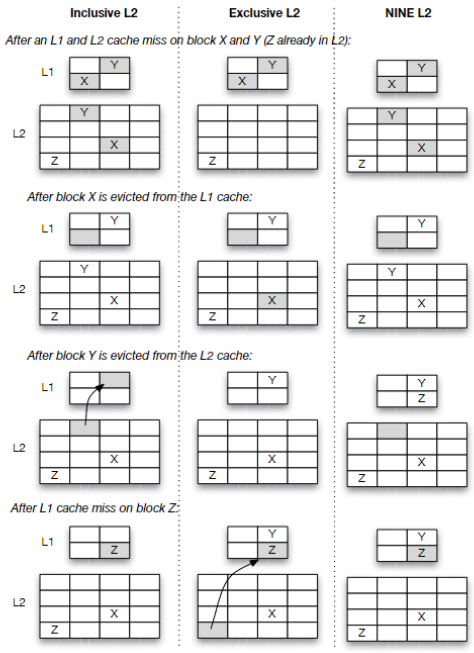
Which kind(s) of caches move a block from one level to the other?

Which kind(s) of caches propagate up an eviction from the L2 to the L1?

Which kind(s) of caches have to inform the L2 about a write to the L1?

In an inclusive cache, can L2 associativity be greater than L1 associativity?

Find and describe the typo in this diagram.



Replacement policies

LRU is a good strategy for cache replacement.

In a set-associative cache, LRU is reasonably cheap to implement. Why?

With the LRU algorithm, the lines can be arranged in an *LRU stack*, in order of recency of reference. Suppose a string of references is—

a b c d a b e a b c d e

and there are 4 lines. Then the LRU stacks after each reference are—

a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*	*e*
	a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*
		a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*
			a	*b*	*c*	*d*	*d*	*d*	*e*	*a*	*b*
*	*	*	*			*			*	*	*

Notice that at each step:

- The line that is referenced moves to the top of the LRU stack.
- All lines below that line keep their same position.
- All lines above that line move down by one position.

How many bits per set are required to keep track of LRU status in both of the implementations described in the text?

- Matrix n^2
- Pseudo-LRU $n - 1$

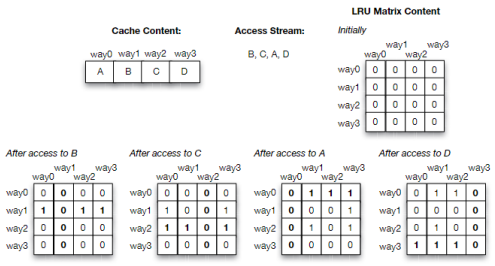


Figure 5.6: Illustrating matrix implementation of the least recently used (LRU) replacement policy.

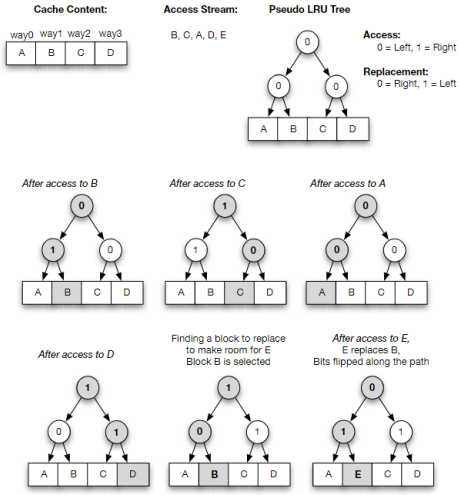


Figure 5.7: Illustration of pseudo-LRU replacement on a 4-way set associative cache.

Performance of coherence protocols

Cache misses have traditionally been classified into four categories:

- *Cold misses* (or "compulsory misses") occur the first time that a block is referenced.
- *Conflict misses* are misses that would not occur if the cache were fully associative with LRU replacement.
- *Capacity misses* occur when the cache size is not sufficient to hold data between references.
- *Coherence misses* are misses caused by the coherence protocol.

The first three types occur in uniprocessors. The last is specific to multiprocessors.

To these, Solihin adds *context-switch* (or "system-related") misses, which are related to task switches.

Let's look at a uniprocessor example, a very small cache that has only four lines.

Let's look first at a fully associative cache, because which kind(s) of misses can't it have?

Here's an example of a reference trace of 0, 2, 4, 0, 2, 4, 6, 8, 0.

Fully associative

	0	2	4	0	2	4	6	8	0
0	0			0				8	
1		2			2				0
2			4			4			
3							6		
	cold	cold	cold	hit	hit	hit	cold	cold	capacity

In a fully associative cache, there are 5 cold misses, because 5 different blocks are referenced.

There are 3 hits.

The remaining reference (the third one to block 0) is not a cold miss.

It must be a capacity miss, because the cache doesn't have room to hold all five blocks.

We'll assume that replacement is LRU; in this case, block 0 replaces the LRU line, which at that point is line 1.

Now let's suppose the cache is 2-way set associative. This means there are two sets, one (set 0) that will hold the even-numbered blocks, and one (set 1) that will hold the odd-numbered blocks.

2-way set-associative

	0	2	4	0	2	4	6	8	0
0	0		4		2		6		0
1		2		0		4		8	
2									
3									
	cold	cold	cold	conflict	conflict	conflict	cold	cold	capacity

Since only even-numbered blocks are referenced in this trace, they will all map to set 0.

This time, though, there won't be any hits.

Classify each of these references as a hit or a particular kind of miss.

References that would have been hits in a fully associative cache, but are misses in a less-associative cache, are conflict misses.

Finally, let's look at a direct-mapped cache. Blocks with numbers congruent to 0 mod 4 map to line 0; blocks with numbers congruent to 1 mod 4 map to line 1, etc.

Direct mapped

	0	2	4	0	2	4	6	8	0
0	0		4	0		4		8	0
1									
2		2			2		6		
3									
	cold	cold	cold	conflict	hit	conflict	cold	cold	capacity

Classify each of these references as a hit or a particular kind of miss.

Of the three conflict misses in the set-associative cache, one is a hit here. Block 2 is still in the cache the second time it is referenced. The other two are conflict misses in this cache.

Now, let's talk about coherence misses.

Coherence misses can be divided into those caused by *true sharing* and those caused by *false sharing* (see p. 236 of the Solihin text).

- False-sharing misses are those caused by having a line size larger than one word. Can you explain?
- True-sharing misses, on the other hand, occur when
  - a processor writes into a cache line, invalidating a copy of the same block in another processor's cache,
  - after which the first processor again references the word that was written to.

How can we attack each of the four kinds of misses?

- To reduce capacity misses, we can
- To reduce conflict misses, we can
- To reduce cold misses, we can
- To reduce coherence misses, we can change the line size

Similarly, context-switch misses can be divided into categories.

- *Replaced* misses are blocks that were replaced while the other process(es) were active.
- *Reordered* misses are blocks that were shoved so far down the LRU stack by the other process(es) that they are replaced soon afterwards (when they otherwise would've stayed in the cache).

Which protocol is best? What cache line size is performs best? What kind of misses predominate?

Simulations

Questions like these can be answered by simulation. Getting the answer right is part art and part science.

Parameters need to be chosen for the simulator. Culler & Singh (1998) selected a single-level 4-way set-associative 1 MB cache with 64-byte lines.

The simulation assumes an idealized memory model, which assumes that references take constant time. Why is this not realistic?

The simulated workload consists of

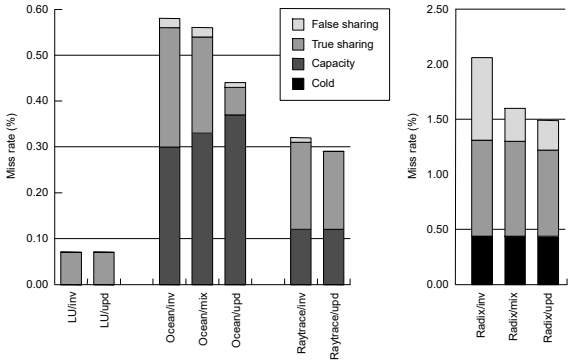
- six parallel programs (Barnes, LU, Ocean, Radix, Radiosity, Raytrace) from the SPLASH-2 suite and
- one multiprogrammed workload, consisting of mainly serial programs.

Invalidate vs. update

with respect to miss rate

Which is better, an update or an invalidation protocol?

Let's look at real programs.



Where there are many coherence misses, update performs better.

If there were many capacity misses, update hurts, because it keeps blocks in my cache even if it's been a long time since *I* referenced them.

with respect to bus traffic

Compare the

- upgrades in inv. protocol

with the

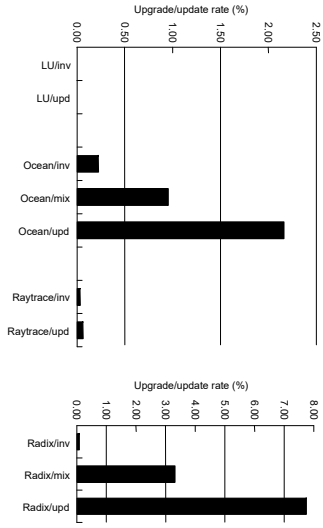
- updates in upd. protocol

Each of these operations produces bus traffic.

Which are more frequent?

Which protocol causes more bus traffic?

The main problem is that one processor tends to write a block multiple times before another processor reads it.



This causes several bus transactions instead of one, as there would be in an invalidation protocol.

Effect of cache line size

on miss rate

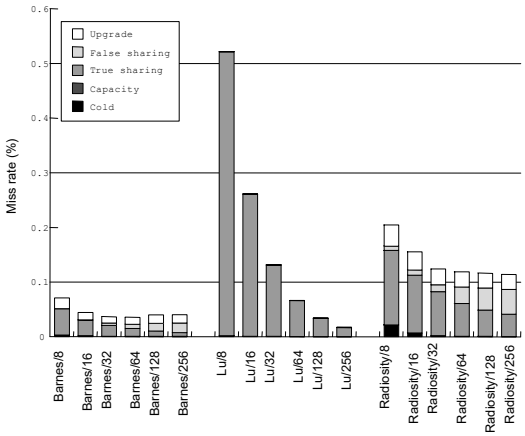
If we increase the line size, what happens to each of the following classes of misses?

- cold misses?
- conflict misses?
- true-sharing misses?

- false-sharing misses?

If we increase the line size, what happens to bus traffic? Increase, because more data is brought in on each miss.

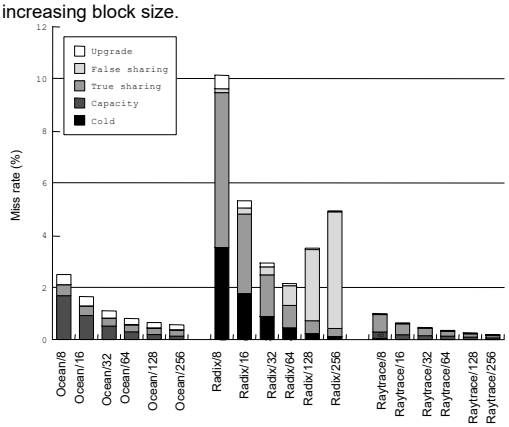
So it is not clear which line size will work best.



Results for the first three applications seem to show that which line size is best? 64 to 256 seems best

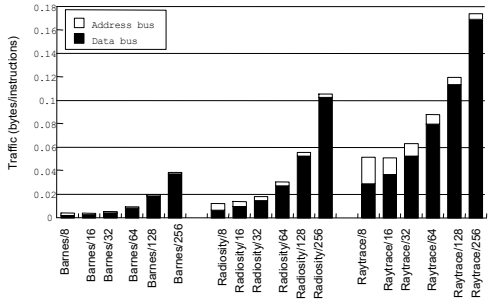
For the second set of applications, which do not fit in cache, Radix shows a greatly increasing number of false-sharing misses with

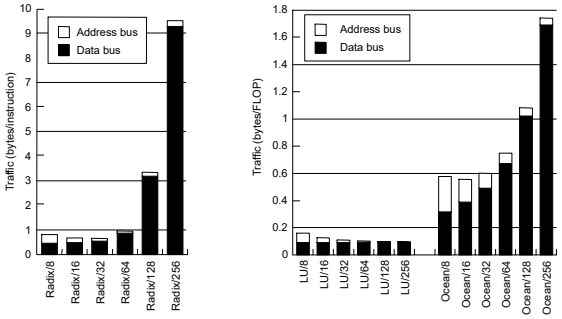
increasing block size.



on bus traffic

Larger line sizes generate more bus traffic.





The results are different than for miss rate—traffic almost always increases with increasing line size.

But address-bus traffic moves in the opposite direction from data-bus traffic.

With this in mind, which line size appears to be best? about 32

Context-switch misses

As cache size gets larger, there are fewer uniprocessor ("natural") cache misses.

But the number of context-switch misses may go up (mcf, soplex) or down (namd, perlbench).

- Why could it go up?
- Why could it go down?

Reordered misses also decline as the cache becomes large. Why?

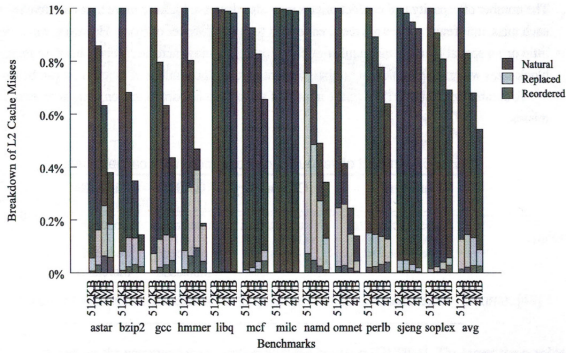


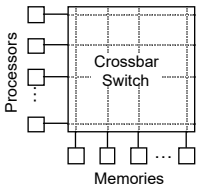
Figure 5.13: Breakdown of the types of L2 cache misses suffered by SPEC2006 applications with various cache sizes. Source: [39].

Physical cache organization

[Solihin §5.6] A cache is *centralized* ("united") if its banks are adjacent on the chip.

What are some advantages of a centralized structure?

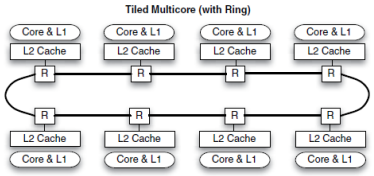
- Uniform access time
- Interconnect between the cache and the next level (e.g., on-chip memory controller) can all be in one place, which simpifies it.



A centralized cache usually uses a crossbar (see also p. 167 of the text).

A cache is *distributed* if its banks are scattered around the chip.

Usually, a portion of the L2 is placed near each L1; this is a *tiled* arrangement.



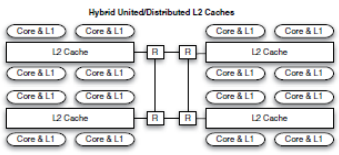
What are some advantages of a distributed structure?

- In replication: A single tile (core, L1 caches, 1 bank of L2) can be designed, & stamped as many times as needed. So it is more scalable, easier to verify, use in next generation (same advs. as multicore!)
- In layout: More feasible for a manycore processor, where wire length and thermal considerations prevent a cache from being centralized.

Hybrid centralized + distributed structure: There's a tradeoff between centralized and distributed.

- A large cache is uniformly slow, especially if it needs to handle coherence.
- A distributed cache requires a lot of interconnections, and routing latency is high if the cache is in too many places.

A compromise is to have an L2 cache that is distributed, but not as distributed as the L1 caches.

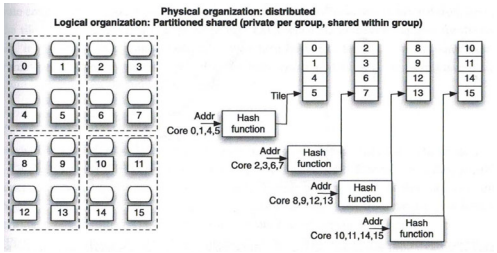


Logical cache organization

[Solihin §5.7] Regardless of whether a cache is centralized or distributed, there are several options in mapping addresses to tiles.

- A processor can be limited to accessing a single tile, the one closest to it (private cache configuration).
  - A block in the local cache may also exist in other caches; the copies must be kept coherent by a coherence protocol.
- All of the tiles can form a large logical cache. The address of a block completely determines what tile it is found in (shared 1-tile associative).
  - It may require a lot of hops to get from a processor to the cache.
- A block can be mapped to two tiles (shared 2-tile associative).
  - Block numbers are arranged to improve distance locality.
- Or, a block can be allowed to map to any tile (full tile associativity).
  - What is the upside?
  - What is the downside?

Another option is a partitioned shared cache organization.



- Can you tell how many tiles each block can map to?
- Can you tell how many *lines* each block can map to?

- How does coherence play a role?

Lock Implementations

[§8.1] Recall the three kinds of synchronization from Lecture 6:

- Point-to-point
- Lock
- Barrier

Performance metrics for lock implementations

- Uncontended latency
  - Time to acquire a lock when there is no contention
- Traffic
  - Lock acquisition when lock is already locked
  - Lock acquisition when lock is free
  - Lock release
- Fairness
  - Swiftness with which a thread can acquire a lock compared to other threads
- Storage
  - As a function of # of threads/processors

The need for atomicity

This code sequence illustrates the need for atomicity. Explain.

```
void lock (int *lockvar) {
  while (*lockvar == 1) {};   // wait until released
  *lockvar = 1;               // acquire lock
}

void unlock (int *lockvar) {
  *lockvar = 0;
}
```

In assembly language, the sequence looks like this:

```
lock: ld R1, &lockvar     // R1 = lockvar
      bnz R1, lock        // jump to lock if R1 != 0
      sti &lockvar, #1    // lockvar = 1
      ret                 // return to caller
unlock: sti  &lockvar, #0 // lockvar = 0
        ret               // return to caller
```

The `ld`-to-`sti` sequence must be executed atomically:

- The sequence appears to execute in its entirety
- Multiple sequences are serialized

Examples of atomic instructions

- **`test-and-set Rx, M`**
  - read the value stored in memory location **`M`**, test the value against a constant (e.g. 0), and if they match, write the value in register **`Rx`** to the memory location **`M`**.
- **`fetch-and-op M`**
  - read the value stored in memory location **`M`**, perform op to it (e.g., increment, decrement, addition, subtraction), then store the new value to the memory location **`M`**.
- **`exchange Rx, M`**
  - atomically exchange (or swap) the value in memory location **`M`** with the value in register **`Rx`**.
- **`compare-and-swap Rx, Ry, M`**
  - compare the value in memory location **`M`** with the value in register **`Rx`**. If they match, write the value in register **`Ry`** to **`M`**, and copy the value in **`Rx`** to **`Ry`**.

How to ensure one atomic instruction is executed at a time:

1. Reserve the bus until done
   - Other atomic instructions cannot get to the bus

2. Reserve the cache block involved until done
   - Obtain exclusive permission (e.g. "M" in MESI)
   - Reject or delay any invalidation or intervention requests until done
3. Provide the "illusion" of atomicity instead
   - Using load-link/store-conditional (to be discussed later)

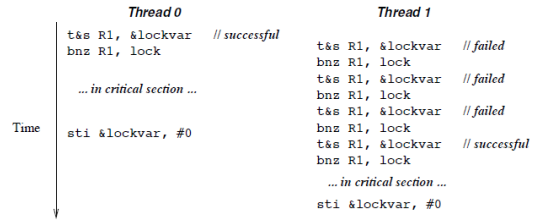
Test and set

`test-and-set` can be used like this to implement a lock:

```
lock:   t&s R1, &lockvar  // R1 = MEM[&lockvar];
                          // if (R1==0) MEM[&lockvar]=1
        bnz R1, lock;     // jump to lock if R1 != 0
        ret               // return to caller
unlock: sti &lockvar, #0  // MEM[&lockvar] = 0
        ret               // return to caller
```

What value does **`lockvar`** have when the lock is acquired? free?
1, 0

Here is an example of **`test-and-set`** execution. Describe what it shows.



Both threads get the lock, but thread 1 tries many times before succeeding.

Let's look at how a sequence of test-and-sets by three processors plays out:

Request	P1	P2	P3	BusRequest
Initially	–	–	–	–
P1: t&s	M	–	–	BusRdX
P2: t&s	I	M	–	BusRdX
P3: t&s	I	I	M	BusRdX
P2: t&s	I	M	I	BusRdX
P1: unlock	M	I	I	BusRdX
P2: t&s	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: t&s	I	I	M	–
P2: unlock	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: unlock	I	I	M	–

How does test-and-set perform on the four metrics listed above?

- Uncontended latency
- Fairness
- Traffic
- Storage

Drawbacks of Test&Set Lock (TSL)

What is the main drawback of test&set locks?

-
-

Without changing the lock mechanism, how can we diminish this overhead?

- Back off: pause for awhile
  - Back off by too little: still a lot of traffic
  - Back off by too much: missed opportunity
- Exponential backoff: Increase the back-off interval exponentially with each failure.

Test and Test&Set Lock (TTSL)

- Busy-wait with ordinary read operations, not test&set.
  - Cached lock variable will be invalidated when release occurs
- When value changes (to 0), try to obtain lock with test&set
  - Only one attempter will succeed; others will fail and start testing again.

Let's compare the code for TSL with TTSL.

TSL:

```
lock:   t&s R1, &lockvar  // R1 = MEM[&lockvar];
                          // if (R1==0) MEM[&lockvar]=1
        bnz R1, lock;     // jump to lock if R1 != 0
        ret               // return to caller
unlock: sti &lockvar, #0  // MEM[&lockvar] = 0
        ret               // return to caller
```

TTSL:

```
lock:   ld R1, &lockvar  // R1 = MEM[&lockvar]
        bnz R1, lock;    // jump to lock if R1 != 0
        t&s R1, &lockvar // R1 = MEM[&lockvar];
                         // if (R1==0)MEM[&lockvar]=1
        bnz R1, lock;    // jump to lock if R1 != 0
        ret              // return to caller

unlock: sti &lockvar, #0 // MEM[&lockvar] = 0
        ret              // return to caller
```


The **lock** method now contains two loops. What would happen if we removed the second loop? Incorrect; two processes could "lock" the lock concurrently.

Here's a trace of a TSL, and then TTSL, execution. Let's compare them line by line.

Fill out this table:

	TSL	TTSL
# BusReads	0	6
# BusReadXs	9	0
# BusUpgrs	0	4
# invalidations	8	5

(What's the proper way to count invalidations?)

TSL: Request	P1	P2	P3	BusRequest
Initially	–	–	–	–
P1: t&s	M	–	–	BusRdX
P2: t&s	I	M	–	BusRdX
P3: t&s	I	I	M	BusRdX
P2: t&s	I	M	I	BusRdX
P1: unlock	M	I	I	BusRdX
P2: t&s	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: t&s	I	I	M	–
P2: unlock	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: unlock	I	I	M	–

TTSL: Request	P1	P2	P3	Bus Request
Initially	–	–	–	–
P1: ld	E	–	-	BusRd
P1: t&s	M	–	–	–
P2: ld	S	S	–	BusRd
P3: ld	S	S	S	BusRd
P2: ld	S	S	S	–
P1: unlock	M	I	I	BusUpgr
P2: ld	S	S	I	BusRd
P2: t&s	I	M	I	BusUpgr
P3: ld	I	S	S	BusRd
P3: ld	I	S	S	–
P2: unlock	I	M	I	BusUpgr
P3: ld	I	S	S	BusRd
P3: t&s	I	I	M	BusUpgr
P3: unlock	I	I	M	–

TSL vs. TTSL summary

- Successful lock acquisition:
  - 2 bus transactions in TTSL
    - 1 BusRd to intervene with a remotely cached block
    - 1 BusUpgr to invalidate all remote copies
  - vs. only 1 in TSL
    - 1 BusRdX to invalidate all remote copies
- Failed lock acquisition:
  - 1 bus transaction in TTSL
    - 1 BusRd to read a copy
    - then, loop until lock becomes free
  - vs. unlimited with TSL
    - Each attempt generates a BusRdX

LL/SC

- TTSL is an improvement over TSL.
- But bus-based locking
  - has a limited applicability (explain) You need a bus!
  - is not scalable with fine-grain locks (explain) Lots more bus traffic. Any lock operation needs to wait for all other lock operations.
- Suppose we could lock a *cache block* instead of a bus …
  - Expensive, must rely on buffering or NACK to prevent a block from being stolen by another processor.
- Instead of providing atomicity, can we provide an illusion of atomicity instead?
  - This would involve detecting a violation of atomicity.
  - If something "happens to" the value loaded, cancel the store (because we must not allow newly stored value to become visible to other processors)
  - Go back and repeat all other instructions (load, branch, etc.).

This can be done with two new instructions:

- Load Linked/Locked (LL)
  - reads a word from memory, and
  - stores the address in a special LL register
  - The LL register is cleared if anything happens that may break atomicity, e.g.,
    - A context switch occurs
    - The block containing the address in the LL register is invalidated.
- Store Conditional (SC)
  - tests whether the address in the LL register matches the store address
  - if so, store succeeds: store goes to cache/memory;
  - else, store fails: the store is canceled, 0 is returned.

Here is the code.

```
lock: LL R1, &lockvar // R1 = lockvar;
                      // LINKREG = &lockvar
      bnz R1, lock    // jump to lock if R1 != 0
      add R1, R1, #1  // R1 = 1
      SC R1, &lockvar // lockvar = R1;
      beqz R1, lock   // jump to lock if SC fails
      ret             // return to caller

unlock: sti &lockvar, #0  // lockvar = 0
        ret               // return to caller
```

Note that this code, like the TTSL code, consists of two loops. Compare each loop with its TTSL counterpart.

- The first loop is identical, except for changing an ld to LL
- The second loop uses an add instruction instead of t&s to set the lock variable to 1. If the LL register is cleared when you try to do a store, you branch back to the top & try again.

Here is a trace of execution. Compare it with TTSL.

Request	P1	P2	P3	BusRequest
Initially	–	–	–	–
P1: LL	E	–	–	BusRd
P1: SC	M	–	–	–
P2: LL	S	S	–	BusRd
P3: LL	S	S	S	BusRd
P2: LL	S	S	S	–
P1: unlock	M	I	I	BusUpgr
P2: LL	S	S	I	BusRd
P2: SC	I	M	I	BusUpgr
P3: LL	I	S	S	BusRd
P3: LL	I	S	S	–
P2: unlock	I	M	I	BusUpgr
P3: LL	I	S	S	BusRd
P3: SC	I	I	M	BusUpgr
P3: unlock	I	I	M	–

- Similar bus traffic
  - Spinning using loads ⇒ no bus transactions when the lock is not free
  - Successful lock acquisition involves two bus transactions. What are they?
- But a failed SC does not generate a bus transaction (in TTSL, all test&sets generate bus transactions).
  - Why don't SCs fail often?

Limitations of LL/SC

- Suppose a lock is highly contended by p threads
  - There are $O(p)$ attempts to acquire and release a lock
  - A single release invalidates $O(p)$ caches, causing $O(p)$ subsequent cache misses
  - Hence, each critical section causes $O(p^2)$ network traffic

- Fairness: There is no guarantee that a thread that contends for a lock will eventually acquire it.

These issues can be addressed by two different kinds of locks.

Ticket Lock

- Ensures fairness, but still incurs $O(p^2)$ traffic
- Uses the concept of a "bakery" queue
- A thread attempting to acquire a lock is given a ticket number representing its position in the queue.
- Lock acquisition order follows the queue order.

Implementation:

```
ticketLock_init(int *next_ticket, int *now_serving) {
  *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving) {
  my_ticket = fetch_and_inc(next_ticket);
  while (*now_serving != my_ticket) {};
}

ticketLock_release(int *next_ticket, int *now_serving) {
  *now_serving++;
}
```

Trace:

Steps	next_ticket	now_serving	my_ticket		
			P1	P2	P3
Initially	0	0	–	–	–
P1: fetch&inc	1	0	0	–	–
P2: fetch&inc	2	0	0	1	–
P3: fetch&inc	3	0	0	1	2
P1:now_serving++	3	1	0	1	2
P2:now_serving++	3	2	0	1	2
P3:now_serving++	3	3	0	1	2

Note that fetch&inc can be implemented with LL/SC.

Array-Based Queueing Locks

With a ticket lock, a release still invalidates $O(p)$ caches.

Idea: Avoid this by letting each thread wait for a unique variable. Waiting processes poll on different locations in an array of size p.

Just change **`now_serving`** to an array! (renamed "**`can_serve`**").

A thread attempting to acquire a lock is given a ticket number in the queue.

Lock acquisition order follows the queue order

- Acquire
  - fetch&inc obtains the address on which to spin (the next array element).
  - We must ensure that these addresses are in different cache lines or memories
- Release
  - Set next location in array to 1, thus waking up process spinning on it.

Advantages and disadvantages:

- $O(1)$ traffic per acquire with coherent caches
  - And each release invalidates only one cache.
- FIFO ordering, as in ticket lock, ensuring fairness
- But, $O(p)$ space per lock
- Good scalability for bus-based machines

Implementation:

```
ABQL_init(int *next_ticket, int *can_serve) {
  *next_ticket = 0;
  for (i=1; i<MAXSIZE; i++)
     can_serve[i] = 0;
  can_serve[0] = 1;
}

ABQL_acquire(int *next_ticket, int *can_serve) {
  *my_ticket = fetch_and_inc(next_ticket) % MAXSIZE;
```

```
  while (can_serve[*my_ticket] != 1) {};
}

ABQL_release(int *next_ticket, int *can_serve) {
  can_serve[*my_ticket + 1] = 1;
  can_serve[*my_ticket] = 0; // prepare for next time
}
```

Trace:

Steps	next_ticket	can_serve[]	my_ticket		
			P1	P2	P3
Initially	0	[1, 0, 0, 0]	–	–	–
P1: f&i	1	[1, 0, 0, 0]	0	–	–
P2: f&i	2	[1, 0, 0, 0]	0	1	–
P3: f&i	3	[1, 0, 0, 0]	0	1	2
P1: can_serve[1]=1	3	[0, 1, 0, 0]	0	1	2
P2: can_serve[2]=1	3	[0, 0, 1, 0]	0	1	2
P3: can_serve[3]=1	3	[0, 0, 0, 1]	0	1	2

Let's compare array-based queueing locks with ticket locks.

Fill out this table, assuming that 10 threads are competing:

	Ticket locks	Array-based queueing locks
#of invalidations		
# of subsequent cache misses		

Comparison of lock implementations

Criterion	TSL	TTSL	LL/SC	Ticket	ABQL
Uncontested latency	Lowest	Lower	Lower	Higher	Higher
1 release max traffic	$O(p)$	$O(p)$	$O(p)$	$O(p)$	$O(1)$
Wait traffic	High	Low	–	–	–
Storage	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(p)$
Fairness guaranteed?	No	No	No	Yes	Yes

Discussion:

- Design must balance latency vs. scalability
  - ABQL is not necessarily best.
  - Often LL/SC locks perform very well.
  - Scalable programs rarely use highly-contended locks.
- Fairness sounds good in theory, but
  - Must ensure that the current/next lock holder does not suffer from context switches or any long delay events

Barriers

[§8.2] Like locks, barriers can be implemented in different ways, depending upon how important efficiency is.

- Performance criteria
  - Latency: time spent from reaching the barrier to leaving it
  - Traffic: number of bytes communicated as a function of number of processors
- In current systems, barriers are typically implemented in software using locks, flags, counters.
  - Adequate for small systems
  - Not scalable for large systems

A thread might have this general organization:

```
..
parallel region
BARRIER
parallel region
BARRIER
..
```

Note that barriers are usually constructed using locks, and thus can use any of the lock implementations in the previous lecture.

A barrier can be implemented like this (first attempt):

```
// shared variables used in barrier & their initial values
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// barrier implementation
void barrier () {
   lock(&barLock);
      if (numArrived == 0)  // first thread sets flag
         canGo = 0;
      numArrived++;
```



```
      int myCount = numArrived;
   unlock(&barLock);

   if (myCount < NUM_THREADS) {
      while (canGo == 0) {}; // wait for last thread
   }
   else { // this is the last thread to arrive
      numArrived = 0; // reset for next barrier
      canGo = 1; // release all threads
 }
}
```

What's wrong with this? When the last thread sets canGo to 1, then it may loop around and hit the barrier again, and then sets canGo back to 0. If any other thread hasn't cleared the barrier by then, it never will (until the next time the barrier is passed).

Sense-reversal centralized barrier

[§8.2.1] The simplest solution to the correctness problem above just toggles the barrier …

- the first time, the threads wait for `canGo` to become 1;
- the next time they wait for it to become 0;
- and then they alternate waiting for it to become 1 and 0 at successive barriers.

Here is the code:

```
// variables used in a barrier and their initial values
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// thread-private variable
int valueToAwait = 0;

// barrier implementation
void barrier () {
   valueToAwait = 1 - valueToAwait; // toggle it
   lock(&barLock);
      numArrived++;
      int myCount = numArrived;
   unlock(&barLock);
```



```
   if (myCount < NUM_THREADS) {
      while (canGo != valueToAwait) {}; // await last thread
   }
   else  { // this is the last thread to arrive
      numArrived = 0; // reset for next barrier
      canGo = valueToAwait; // release all threads
   }
}
```

How does the traffic at this barrier scale?

Combining-tree barrier

[§8.2.2] A tree-based strategy can be used to reduce contention, similarly to the way we used partial sums in Lecture 6.

- Threads represent the leaf nodes of a tree.
- The non-leaf nodes are the variables that the threads spin on.
- Each thread spins on the variable of its immediate parent, which constitutes an intermediate barrier.
- Once all threads have arrived at the intermediate barrier, one of these threads goes on and spins on the variable immediately above.
- This is repeated until the root is reached. At this point, the root releases all threads by setting a flag.

How does this improve performance?

But there is an offsetting cost to a combining tree. What is it?

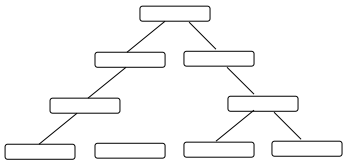
[§8.2.3] In very large supercomputers, however, this technique does not suffice.

The BlueGene/L system has a special *barrier network* for implementing barriers and broadcasting notifications to processors.

The network contains four independent channels.

Each level does a global **and** of the signals from the levels below it.

The signals are combined in hardware and propagate to the top of a combining tree.



The tree can also be used to do a global interrupt when the entire machine or partition must be stopped as soon as possible "for diagnostic purposes."

In this case, each level does a global **or** of the signals from beneath.

Once the signal propagates to the top of the tree, the resultant notification is broadcast down the tree.

The round-trip latency is only 1.5 μs for a system of 64K nodes.

Cache Coherence vs. Memory Consistency

- Cache coherence
  - deals with ordering of writes to a **single** memory location
  - only needed for systems with caches
- Memory consistency
  - deals with ordering of reads/writes to *all* memory locations
  - needed in systems with or without caches

Why is a memory consistency model needed?

[§9.1] Programmer's intuition:

P0:	P1:
S1: datum = 5;	S3: while (!datumIsReady);
S2: datumIsReady = 1;	S4: … = datum

Programmers expect **S4** to read the new value of **datum** (i.e., **5**).

This expectation is violated if—

- **S2** appears to be executed before **S1**
- **S4** appears to be executed before **S3**

Thus, *Hypothesis 1: Program-order expectation*

Programmers expect memory accesses in a thread to be executed in the same order in which they occur in the source code.

Not only the executing thread, but *all* threads, are expected to see them in this order.

P0:	P1:	P2:
S1: x = 5; S2: xReady = 1;	S3: while (!xReady) {}; S4: y = x + 4; S5: xyReady = 1;	S6: while (!xyReady) {}; S7: z = x * y;

Let's say, initially, **x = y = z = xReady = xyReady = 0**

As a programmer, what would you expect to be the value of **z** at **S7**?

This implies that if the new value of **x** has been propagated to **P2**, it has also been propagated to **P1.**

Thus, *Hypothesis 2: Atomicity expectation*

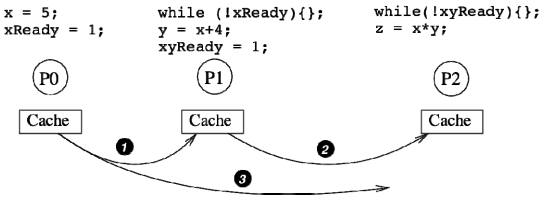
A read or write happens instantaneously with respect to all processors.

How can the atomicity expectation be violated?

Step 1: New values of **x** and **xReady** have been propagated to **P1**, but have not reached **P2**.

Step 2: New values of **y** and **xyReady** have been propagated to **P2** before **x** is propagated to **P2**.

Step 3: When **x** is propagated to **P2**, **P2** has already read the old value of **x**, and **z** has been set to **0**.



Is there any other way that a violation of store atomicity can lead to a wrong value for **z**?

What is *another* "incorrect" value that could be written for **z**? Explain how this could happen.

Summary of programmer's expectations:

Memory accesses emanating from a processor should be performed in program order, and each of them should be performed atomically.

These expectations were incorporated in Lamport's 1979 definition of sequential consistency:

> A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

Sequentially consistent vs. non-SC outcomes

Consider these code sequences, with **a** and **b** initialized to **0**.

P0:	P1:
S1: a = 1;	S3: print b;
S2: b = 1;	S4: print a;

Note that this program is *non-deterministic* due to a lack of synchronization.

Under SC, **S1** → **S2** and **S3** → **S4** are guaranteed

Assuming SC, what values might possibly be printed for **a** and **b**?

S1, S2, S3, S4 ➔ a = 1, b = 1

S1, S3, S2, S4 ➔ a = 1, b = 0

S1, S3, S4, S2 ➔ a = 1, b = 0

S3, S4, S1, S2 ➔ a = 0, b = 0

What values for **a**, **b** are impossible? a = 0, b = 1

Prove it.

For **a** to print as **0**, it must be that **S4** → **S1**: e.g.,

For **b** to print as **1**, it must be that **S2** → **S3**: e.g.,

Both of these conditions cannot hold. Prove it.

On a non-SC machine, the outcome of **a**, **b** = **0**, **1** is possible. What statement ordering can produce it? S4, S1, S2, S3

In this case, which of the two SC precedence guarantees (above) is violated? Program order: S4 → S3

Let's take another example.

P0:	P1:
S1: a = 1;	S3: b = 1;
S2: print b;	S4: print a;

Exercise: Assuming that **a** and **b** are initialized to 0,

- what values can be printed under SC?
- what values are impossible to print under SC?
- prove that the impossible results can only occur if SC is violated.

Answer: Note that the program is non-deterministic due to a lack of synchronization.

With SC, **S1** → **S2** and **S3** → **S4** are guaranteed

a prints as 0 ➔ S4 → S1
b prints as 0 ➔ S2 → S3

Program order ➔ S1 → S2 and S3 → S4

So now we have S1 → S2 → S3 → S4 → S1, which is a contradiction.

On a nondeterministic machine, the outcome **a**, **b** = 0, 0 is possible.

- **S4**, **S1**, **S2**, **S3**
  - In this case, **S3** → **S4** is violated
- **S2**, **S3**, **S4**, **S1**

- In this case, **S1** → **S2** is violated

Both of the previous examples are non-deterministic.

Non-deterministic codes are notoriously hard to debug.

But non-determinism may have legitimate uses. See Code 3.16 (ocean-current simulation) and 3.18 (smoothing filter for grayscale image).

So, does preserving ordering of memory accesses matter?

- Probably not if non-determinism is intentional
- Otherwise, yes, because:
  - Helps keep programmers sane during debugging.
  - Even properly synchronized programs need ordering for the synchronization to work properly.

Building a SC system

[§9.2] Which of the two hypotheses (expectations) can be guaranteed by software? Program order

- Ensure that compiler does not reorder memory accesses;
- Declare critical variables as volatile (to avoid register allocation, code elimination, etc.)

What hypothesis needs to be maintained by hardware? Atomicity

- Execute one memory access one at a time, in program order. One access needs to be complete before the next can start.
- In the processor pipeline, memory accesses can be overlapped or reordered.
  - But they must go to the cache in program order.
  - A load is complete when the block has been read from the cache.

  - A store is complete when an invalidation has been posted (on a bus) or acknowledged (see details in §10.2.1).

Example of SC Ordering

```
S1: ld R1, A
S2: ld R2, B
S3: st R3, C
S4: st R4, D
S5: st R5, D
```

S1 must complete before **S2**, **S2** before **S3**, etc.

Implications

- If **S1** is a cache miss but **S2** is a cache hit, **S2** still must wait until **S1** is completed. Same with **S3** and **S4**.
- **S4** must wait for **S3** to complete, even though stores are often retired early.
- **S5** must wait for **S4** to complete, even though they are to the same location!

Improving SC performance

Via prefetching

We still have to obey ordering, but we can make each load/store complete faster, e.g. by converting cache misses into cache hits:

- Employ load prefetching
  - As soon as address is known/predictable,
    - fetch before previous loads have completed,
  - issue a prefetch request to fetch the block in Exclusive/Shared state
- Employ store prefetching
  - As soon as address is known/predictable, issue a prefetch request to fetch the block in Modified state

But this is not a perfect strategy. Why not?

- Prefetch too late ⇒
- Prefetch too early ⇒

Via speculation

We can violate ordering, but undo the effect if atomicity is violated.

- The ability to undo execution and re-execute is already present in out-of-order processors (as covered in ECE 463/563).
  - So, we only need to determine when atomicity has been violated.
- Consider load A, followed by load B
  - In strict SC, load B must wait until load A completes
  - With speculation, load B accesses the cache anyway; the processor just marks load B as speculative
  - If B is invalidated before it "retires," atomicity has been violated.
  - In this case, the architecture cancels B and re-executes it.

Store speculation is harder, because stores cannot be canceled. Hence, only load speculation is employed.

Relaxed Memory-Consistency Models

Review. Why are relaxed memory-consistency models needed?

How do relaxed MC models require programs to be changed?

The "safety net" between operations whose order needs to be guaranteed is often a *fence* instruction.

- The fence ensures that memory operations that are "younger" are not issued until the older mem ops have globally performed. The newer instruction must
  - wait until all older writes have been posted on the bus (or received InvAck);
  - wait until all older reads have completed;
  - flush the pipeline to avoid issuing younger mem ops early

- Programmers must insert fences.

What if amateur programmers perform their own synchronization, and forget fences?

A continuum of consistency models

Sequential consistency is one view of what a programming model should guarantee.

Let us introduce a way of diagramming consistency models. Suppose that—

- The value of a particular memory word in processor 2's local memory is 0.
- Then processor 1 writes the value 1 to that word of memory. Note that this is a remote write.
- Processor 2 then reads the word. But, being local, the read occurs quickly, and the value 0 is returned.

What's wrong with this? It looks like processor 2 retrieved an old value.

This situation can be diagrammed like this (the horizontal axis represents time):

P_1:	$W(x)1$	
P_2:		$R(x)0$

Depending upon how the program is written, it may or may not be able to tolerate a situation like this.

But, in any case, the programmer must understand what can happen when memory is accessed in a DSM system.

Sequential consistency

Sequential consistency: *The result of any execution is the same as if*

- *the memory operations of all processors were executed in some sequential order, and*
- *the operations of each individual processor appear in this sequence in the order specified by its program.*

Sequential consistency does *not* mean that writes are instantly visible throughout the system (it would be impossible to implement that anyway).

The example below illustrates two sequentially consistent executions.

Note that a read from P_2 is allowed to return an out-of-date value (because it has not yet "seen" the previous write).

P_1:	$W(x)1$		
P_2:		$R(x)0$	$R(x)1$

P_1:	$W(x)1$		
P_2:		$R(x)1$	$R(x)1$

From this we can see that running the same program twice in a row in a system with sequential consistency may not give the same results.

Causal consistency

The first step in weakening the consistency constraints is to distinguish between events that are potentially *causally* connected and those that are not.

Two events are causally related if one can influence the other.

P_1:	$W(x)1$		
P_2:		$R(x)1$	$W(y)2$

Here, the write to x could influence the write to y, because

On the other hand, without the intervening read, the two writes would not have been causally connected:

P_1:	$W(x)1$	
P_2:		$W(y)2$

The following pairs of operations are potentially causally related:

- A read followed by a later write by the same processor.
- A write followed by a later read to the same location.
- The transitive closure of the above two types of pairs of operations.

Operations that are not causally related are said to be *concurrent*.

Causal consistency: *Writes that are potentially causally related must be seen in the same order by all processors.*

Concurrent writes may be seen in a different order by different processors.

Here is a sequence of events that is allowed with a causally consistent memory, but disallowed by a sequentially consistent memory:

P_1:	$W(x)1$		$W(x)3$	
P_2:		$R(x)1$	$W(x)2$	
P_3:		$R(x)1$		$R(x)3$ $R(x)2$
P_4:		$R(x)1$		$R(x)2$ $R(x)3$

Why is this not allowed by sequential consistency?

Why is this allowed by causal consistency?

What is the violation of causal consistency in the sequence below?

P_1:	$W(x)1$			
P_2:		$R(x)1$	$W(x)2$	
P_3:				$R(x)2$ $R(x)1$
P_4:				$R(x)1$ $R(x)2$

The two writes to x are causally connected, so must be seen in the same order.

Without the $R(x)1$ by P_2, this sequence would've been causally consistent.

Implementing causal consistency requires the construction of a dependency graph, showing which operations depend on which other operations.

Processor consistency

Causal consistency requires that all processes see causally related writes from *all* processors in the same order.

The next step is to relax this requirement, to require only that writes from the *same* processor be seen in order. This gives processor consistency.

Processor consistency: *Writes performed by a single processor are received by all other processors in the order in which they were issued.*

Writes from different processors may be seen in a different order by different processors.

~~Processor consistency~~ PRAM consistency would permit this sequence that we saw violated causal consistency:

P_1:	$W(x)1$			
P_2:		$R(x)1$	$W(x)2$	
P_3:				$R(x)2$ $R(x)1$
P_4:				$R(x)1$ $R(x)2$

Another way of looking at this model is that all writes generated by different processors are considered to be concurrent.

Note: Some definitions of processor consistency require cache coherence too. Processor consistency *without* cache coherence is called PRAM consistency.

Exercise: What is the strongest consistency model that each of the following satisfy?

P_1:	$W(x)1$			
P_2:		$R(x)1$	$W(x)2$	
P_3:				$R(x)1$ $R(x)2$
P_4:				$R(x)2$ $R(x)1$

P_1:	$W(y)1$			
P_2:		$R(x)1$	$W(y)2$	
P_3:				$R(y)1$ $R(y)2$
P_4:				$R(y)2$ $R(y)1$

P_1:	$W(x)1$			
P_2:		$R(x)1$	$W(y)2$	
P_3:				$R(x)1$ $R(y)2$
P_4:				$R(y)2$ $R(x)1$

Sometimes processor consistency can lead to counterintuitive results. Assume that *a* and *b* are initialized to 0.

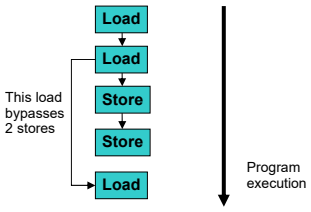
P_1:	P_2:
$a = 1;$	$b = 1;$
if $(b == 0)$	**if** $(a == 0)$
$kill(p_2);$	$kill(p_1);$

At first glance, it seems that no more than one process should be killed.

With processor consistency, however, it is possible for both to be killed. Explain how.

What processor consistency guarantees

- SC ensures ordering of
  - LD → LD
  - LD → ST
  - ST → LD
  - ST → ST
- PC removes the ST→LD constraint, with significant implications for ILP:
  - Values can be loaded into other caches, even if there's a store to the same location in some write buffer.
  - Loads do not wait for stores to complete ("perform"), they access the cache right away (without being speculative!).
  - A load dependent on an older store (in the same processor) can "bypass" (directly obtain the store value before it is stored).
- PC also removes write atomicity.



P1:
data = 2000;
flag = 1;

P2:
while (*flag* == 0) {};
print *data*;

PC produces SC results, because ordering between 2 stores is preserved.

P1:
flag1 = 1;
if (*flag2* == 0)
…

P2:
flag2 = 1;
if (*flag1* == 0)
…

PC fails to produce SC results, because PC does not guarantee ordering betw. store & younger load

- How close is PC to programmers' expectation?
  - Most of the time, very close (e.g., post-wait synchronization works correctly)
  - Major OSes are ported to PC with relative ease
- Cases that cause errors in PC usually are due to races that also happen in SC.
  - However, debugging races in PC is more difficult.

Weak ordering

Processor consistency is still stronger than necessary for many programs, because it requires that writes originating in a single processor be seen in order everywhere.

But it is not always necessary for other processors to see writes in order—or even to see all writes, for that matter.

Suppose a processor is in a tight loop in a critical section, reading and writing variables.

Other processes aren't supposed to touch these variables until the process exits its critical section.

Under processor consistency, the memory has no way of knowing that other processes don't care about these writes, so it has to propagate all writes to all other processors in the normal way.

To relax our consistency model further, we have to divide memory operations into two classes and treat them differently.

- Accesses to *synchronization variables* are sequentially consistent.
- Accesses to other memory locations can be treated as concurrent.

This strategy is known as *weak ordering*.

With weak ordering, we don't need to propagate accesses that occur during a critical section.

We can just wait until the process exits its critical section, and then—

- make sure that the results are propagated throughout the system, and
- stop other actions from taking place until this has happened.

Similarly, when we want to enter a critical section, we need to make sure that all previous writes have finished.

These constraints yield the following definition:

Weak ordering: *A memory system exhibits weak ordering iff—*

1. *Accesses to synchronization variables are sequentially consistent.*
2. *No access to a synchronization variable can be performed until all previous writes have completed everywhere.*
3. *No data access (read or write) can be performed until all previous accesses to synchronization variables have been performed.*

Thus, by doing a synchronization before reading shared data, a process can be assured of getting the most recent values written by other processes before their immediately preceding *S*s.

Note that this model does not allow more than one critical section to execute at a time, even if the critical sections involve disjoint sets of variables.

This model puts a greater burden on the programmer, who must decide which variables are synchronization variables.

Weak ordering says that memory does not have to be kept up to date between synchronization operations.

This is similar to how a compiler can put variables in registers for efficiency's sake. Memory is only up to date when these variables are written back.

If there were any possibility that another process would want to read these variables, they couldn't be kept in registers.

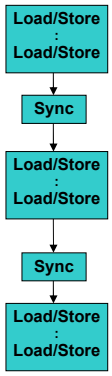
This shows that processes can live with out-of-date values, provided that they know when to access them and when not to.

The following is a legal sequence under weak ordering. Can you explain why?

P_1:	$W(x)1$	$W(x)2$	S	
P_2:		$R(x)2$	$R(x)1$	S
P_3:		$R(x)1$	$R(x)2$	S

Here's a sequence that's illegal under weak ordering. Why?

P_1:	$W(x)1$	$W(x)2$	S	
P_2:		S		$R(x)1$



Synch may be implemented as a lock acquire/release

Before a synch, all previous ops must finish.
Before any ld/st, all previous synch must finish.

Why safe? Typically within a critical section, we have made sure that only one process is inside, thus safe to reorder anything in the critical section.

Outside a critical section, we usually do not care about the order of mem ops (we would have used synchronization if we had cared).

How to know whether a particular ld/st serves as a synchronization point?

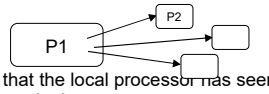
- Assume all atomic instructions are synchronization points
  - fetch-and-op, test-and-set
- Assume all load linked (LL) and store conditional (SC) are synchronization points

Release consistency

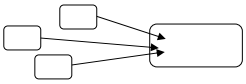
Weak ordering does not distinguish between entry to critical section and exit from it.

Thus, on both occasions, it has to take the actions appropriate to both:

- making sure that all locally initiated writes have been propagated to all other memories, and



- making sure that the local processor has seen all previous writes anywhere in the system.



If the memory could tell the difference between entry and exit of a critical section, it would only need to satisfy one of these conditions.

Release consistency provides two operations:

- *acquire* operations tell the memory system that a critical section is about to be entered.
- *release* operations say a c. s. has just been exited.

It is possible to acquire or release a single synchronization variable, so more than one critical section can be in progress at a time.

When an acquire occurs, the memory will make sure that all the local copies of shared variables are brought up to date.

When a release is done, the shared variables that have been changed are propagated out to the other processors.

But—

- doing an acquire does not guarantee that locally made changes will be propagated out immediately.
- doing a release does not necessarily import changes from other processors.

Here is an example of a valid event sequence for release consistency (*A* stands for "acquire," and *Q* for "release" or "quit"):

P_1:	$A(L)$	$W(x)1$	$W(x)2$	$Q(L)$	
P_2:				$A(L)R(x)2$	$Q(L)$
P_3:					$R(x)1$

Note that since P_3 has not done a synchronize, it does not necessarily get the new value of *x*.

Release consistency: A system is release consistent if it obeys these rules:

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed.*

2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.*
3. *The acquire and release accesses must be processor consistent.*

If these conditions are met, and processes use *acquire* and *release* properly, the results of an execution will be the same as on a sequentially consistent memory.

Summary: Sequential consistency is possible, but costly. The model can be relaxed in various ways.
Consistency models not using synchronization operations:

Type of consistency	Description
Sequential	All processes see all shared accesses in same order.
Causal	All processes see all causally related shared accesses in the same order.
Processor	All processes see writes from each processor in the order they were initiated. Writes from different processors may not be seen in the same order, except that writes to the same location will be seen in the same order everywhere.

Consistency models using synchronization operations:

Type of consistency	Description
Weak	Shared data can only be counted on to be consistent after a synchronization is done.
Release	Shared data are made consistent when a critical region is exited.

The following diagram contrasts various forms of consistency.