Multiprocessors

05-1

Material from Chapter 8 of Hennessy and Patterson.

Outline

- Message Passing and Shared Memory Parallel Computers (Outline only.)
- Cache Coherence (Outline only.)
- Snooping Protocols (Outline only.)
- Directory Protocols
- Synchronization Instructions and Operations
- Consistency Models

Directory-Based Cache Coherence Protocols

05-2

Idea: Memory keeps track of where blocks cached.

At memory, for each block:

- Caches holding block.
  Bit vector with one bit per processor or list of processors.
- State indicating status of black.
  Main states: **Uncached, Shared, Exclusive**

At cache, for each line:

- State.
  Main states: **Invalid, Shared, Exclusive**

Protocol Messages

Sent from cache to memory and *vice versa*.

Protocol Messages

05-3

Sent by one protocol processor through the network . . .

. . . and received by a remote protocol processor.

Receiving protocol processor performs some action and may send a response.

Protocol Message Contents

- Type
- (In some types.) Sending processor.
- Address.
- (In some types.) Data.

Typical Protocol Messages

05-4

Read Miss	*Cache to Memory*	PA

If **Exclusive** fetch data. Return data to cache.

Write Miss	*Cache to Memory*	PA

If **Exclusive** fetch/invalidate data, if **shared** invalidate other copies. Return data to cache.

Invalidate	*Memory to Cache*	A

Change line to invalid.

Fetch	*Memory to Cache*	A

Return data, change state to **shared**.

Fetch/inv.	*Memory to Cache*	A

Return data, change state to **invalid**.

Data value reply	*Memory to Cache*	AD

Write data to cache.

Data write back	*Cache to Memory*	AD

Update memory with data from cache.

Snooping Performance: Number of Processors

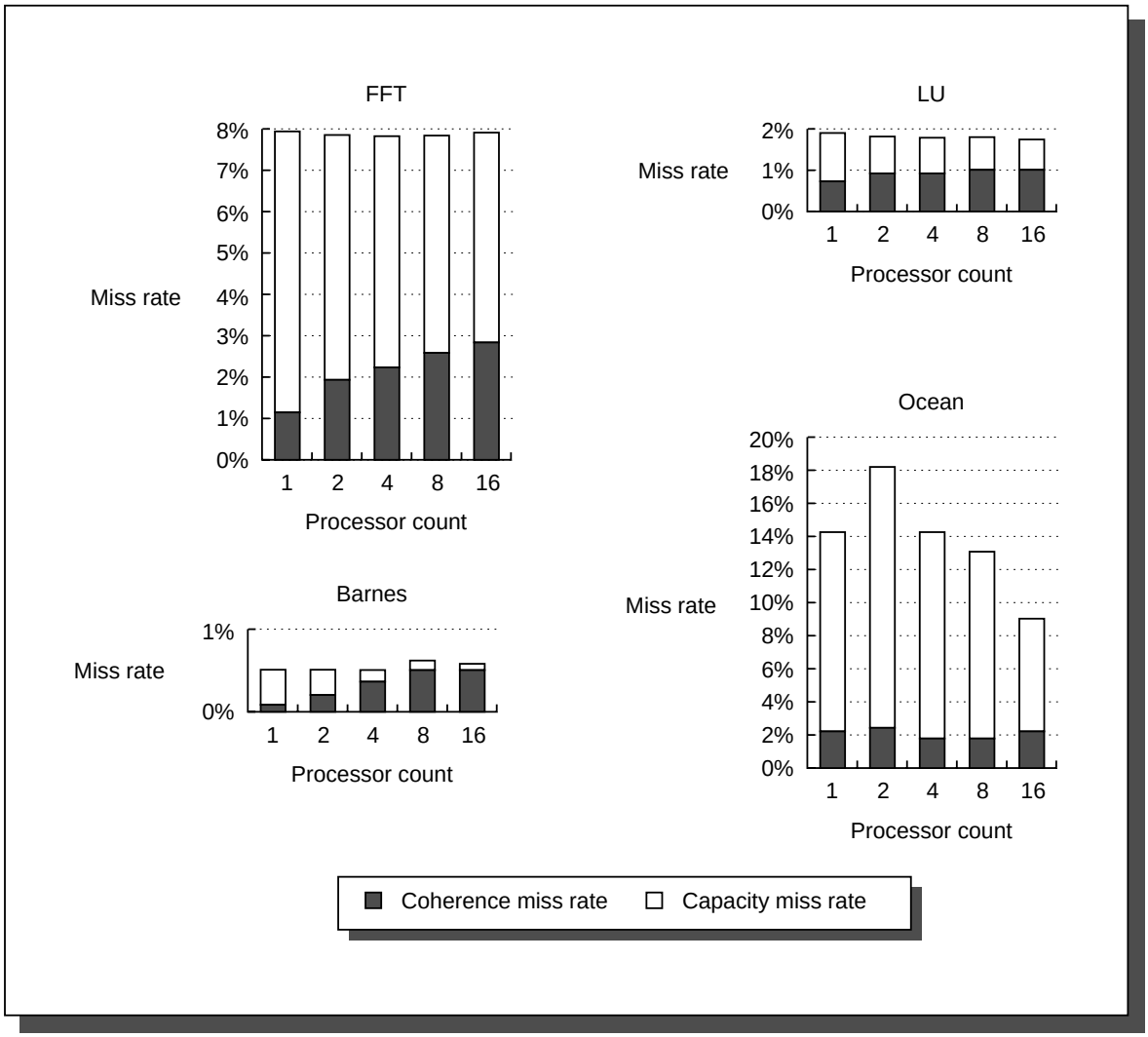


FIGURE 8.13 This data miss rates can vary in nonobvious ways as the processor count is increased from one to 16.

Cache: 64KB 2-way set associative.

Problem size constant.

Miss categorization:

Coherence Miss: Write to shared line.

Capacity Miss: Miss that's not a coherence miss.

EE 7700-4 Lecture Transparency. Formatted 14:48, 24 November 1998 from lsli05.

Snooping Performance: Cache Size

05-6

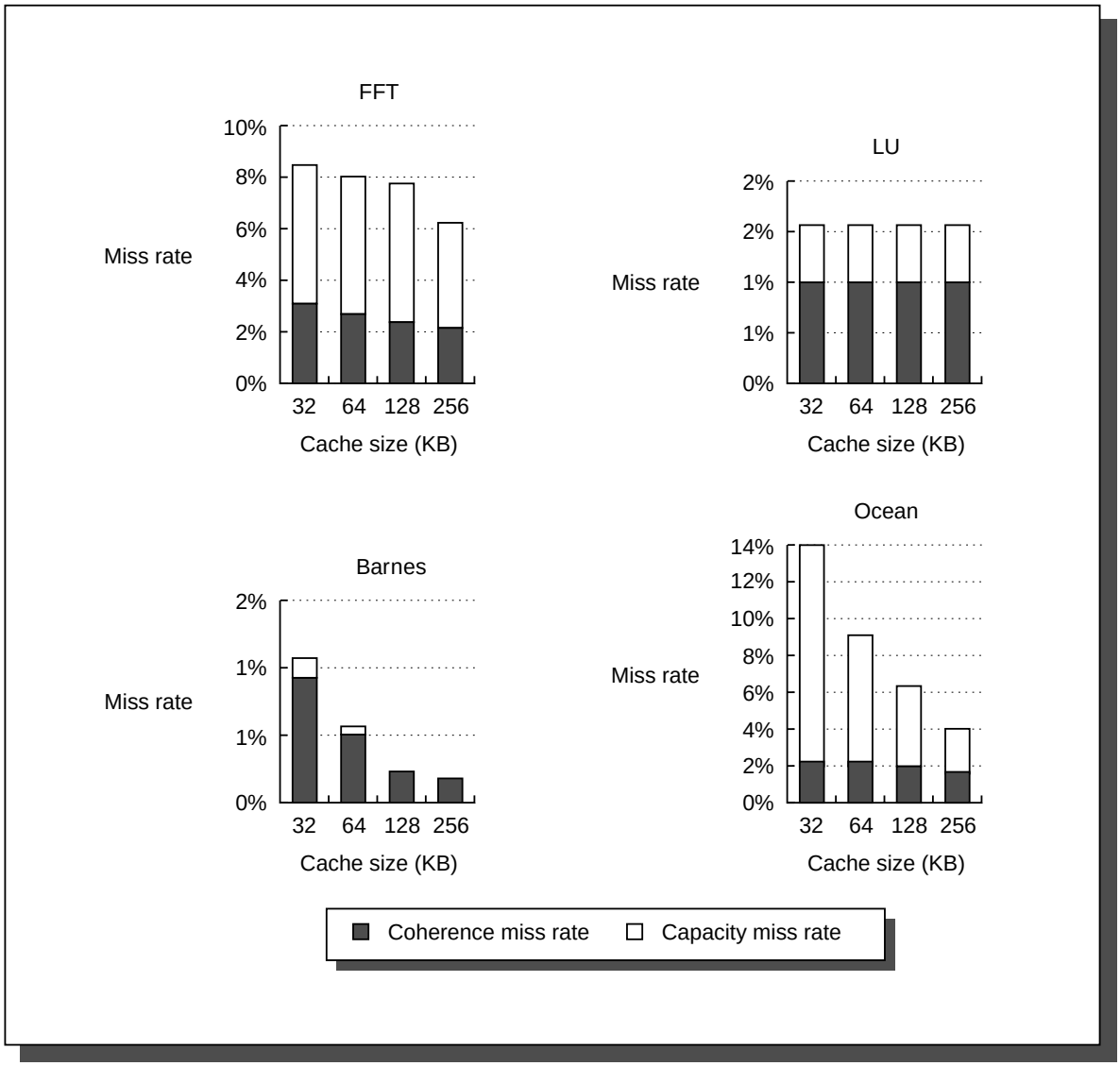


FIGURE 8.14 The miss rate usually drops as the cache size is increased, although coherence misses dampen the effect.

EE 7700-4 Lecture Transparency. Formatted 14:48, 24 November 1998 from lsli05.

05-6

Snooping Performance: Line Size

05-7

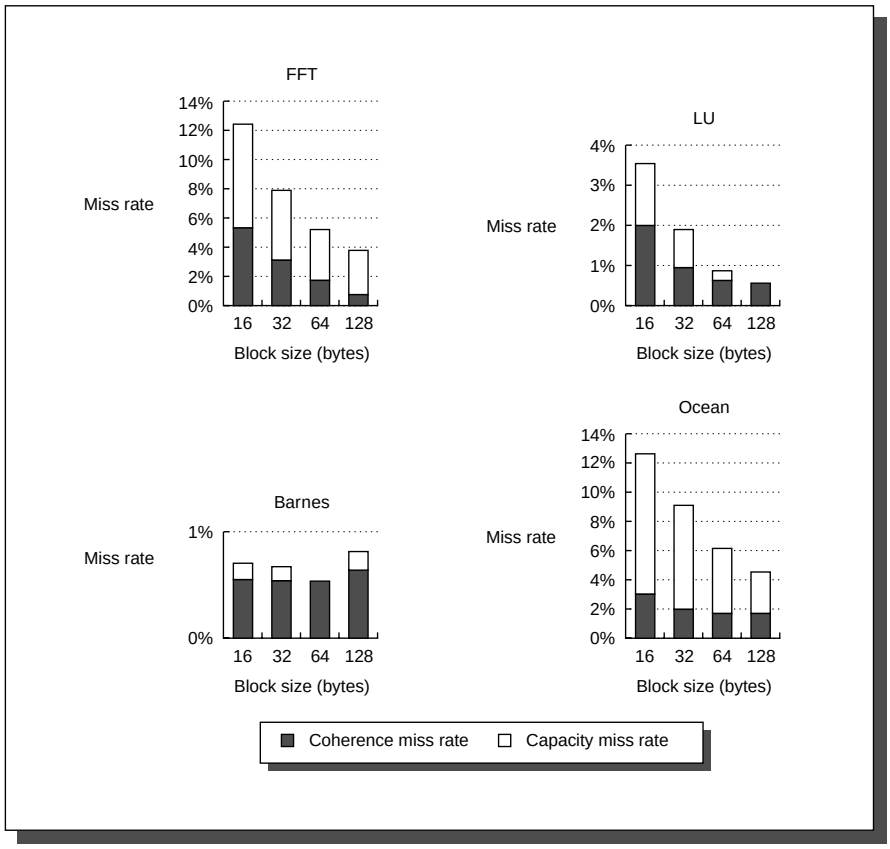


FIGURE 8.15 The data miss rate drops as the block size is increased.

Cache size constant.

Snooping Performance: Line Size

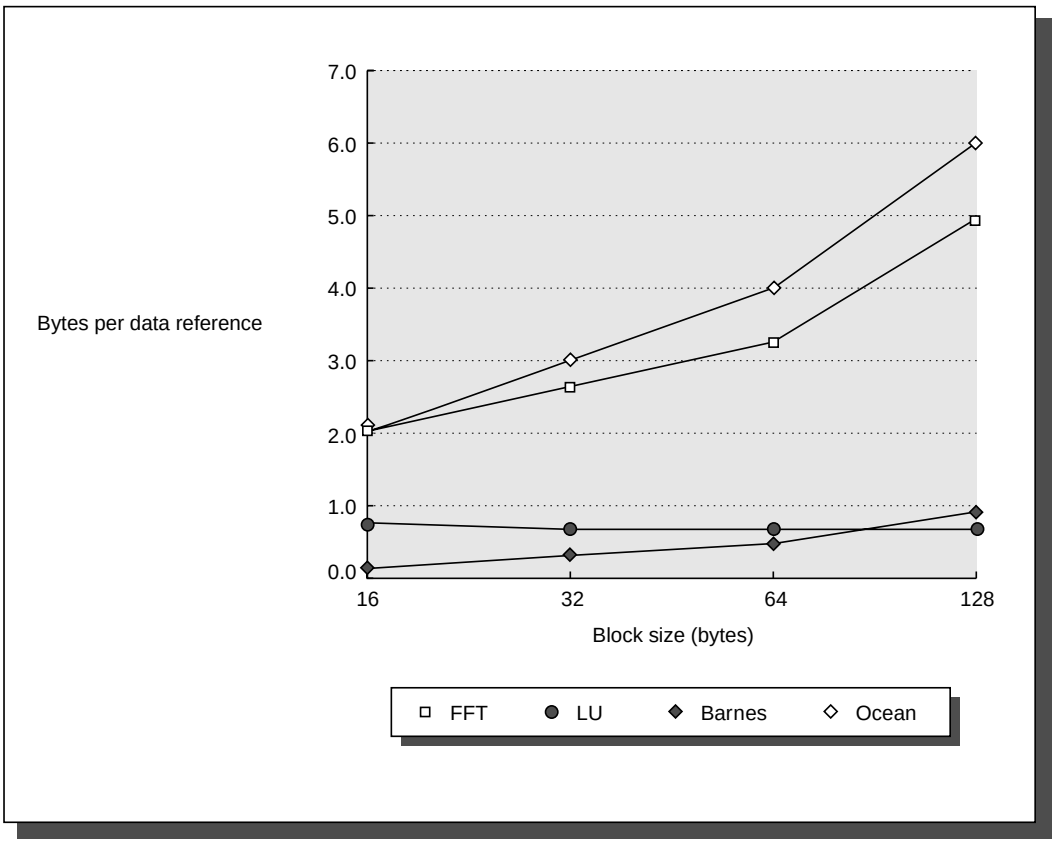


FIGURE 8.16 Bus traffic for data misses climbs steadily as the block size in the data cache is increased.

Timing ignored.

Directory Performance

05-9

Simulated System

128 KB, 64-Byte Line, 2-Way Set Associative Cache

Miss Categorization

Local Miss:
Miss requiring no communication with other nodes.

Remote Miss:
Miss requiring any communication with other nodes.

Miss Significance

Local Miss

Well placed private data.

Remote Miss

True sharing. (Read of data written by another processor.)

False sharing.

Private data kept in a remote memory.

Directory Performance: Processor Count

05-10

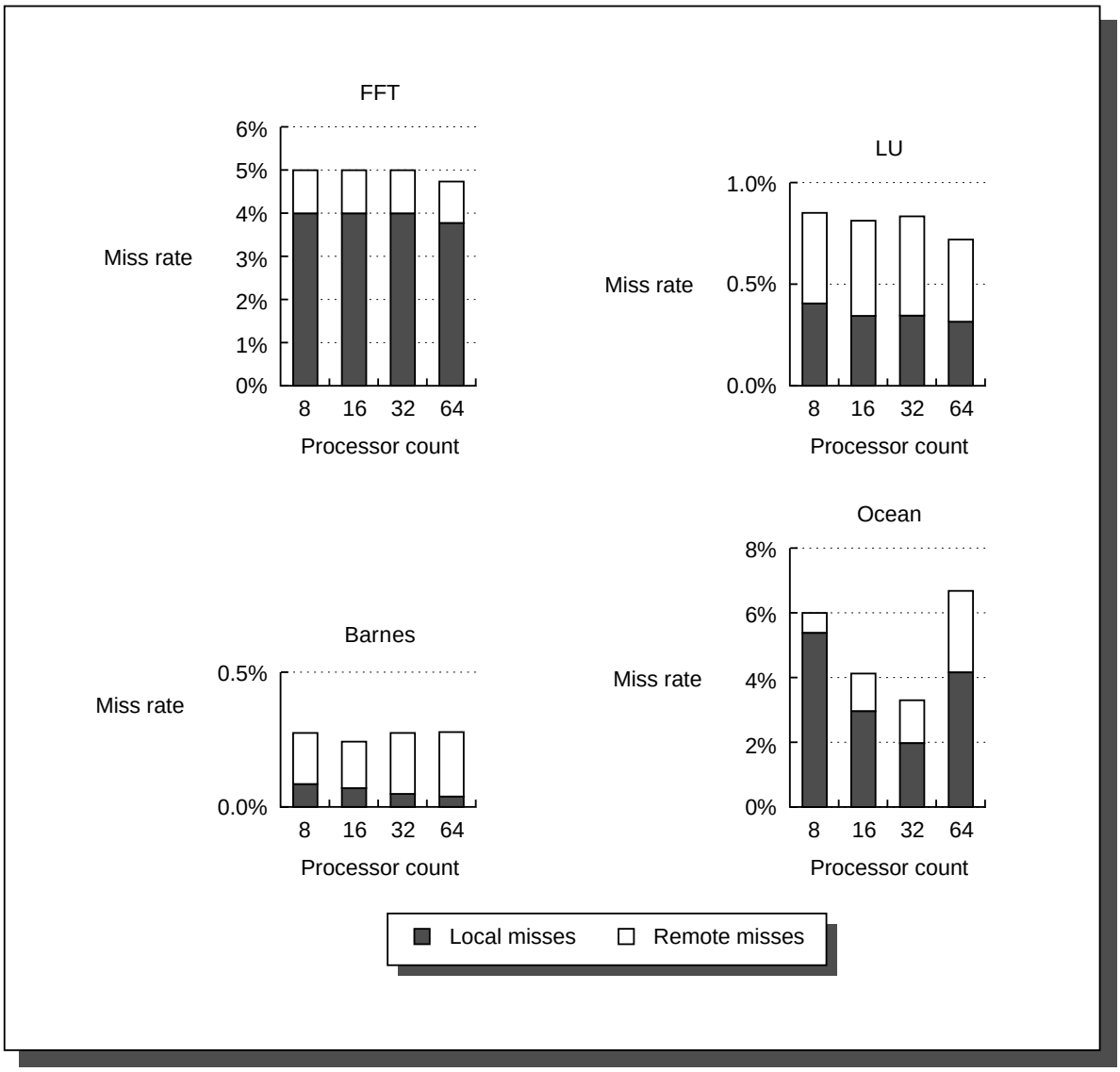


FIGURE 8.26 The data miss rate is often steady as processors are added for these benchmarks.

EE 7700-4 Lecture Transparency. Formatted 14:48, 24 November 1998 from lsli05.

05-10

Directory Performance: Cache Size

05-11

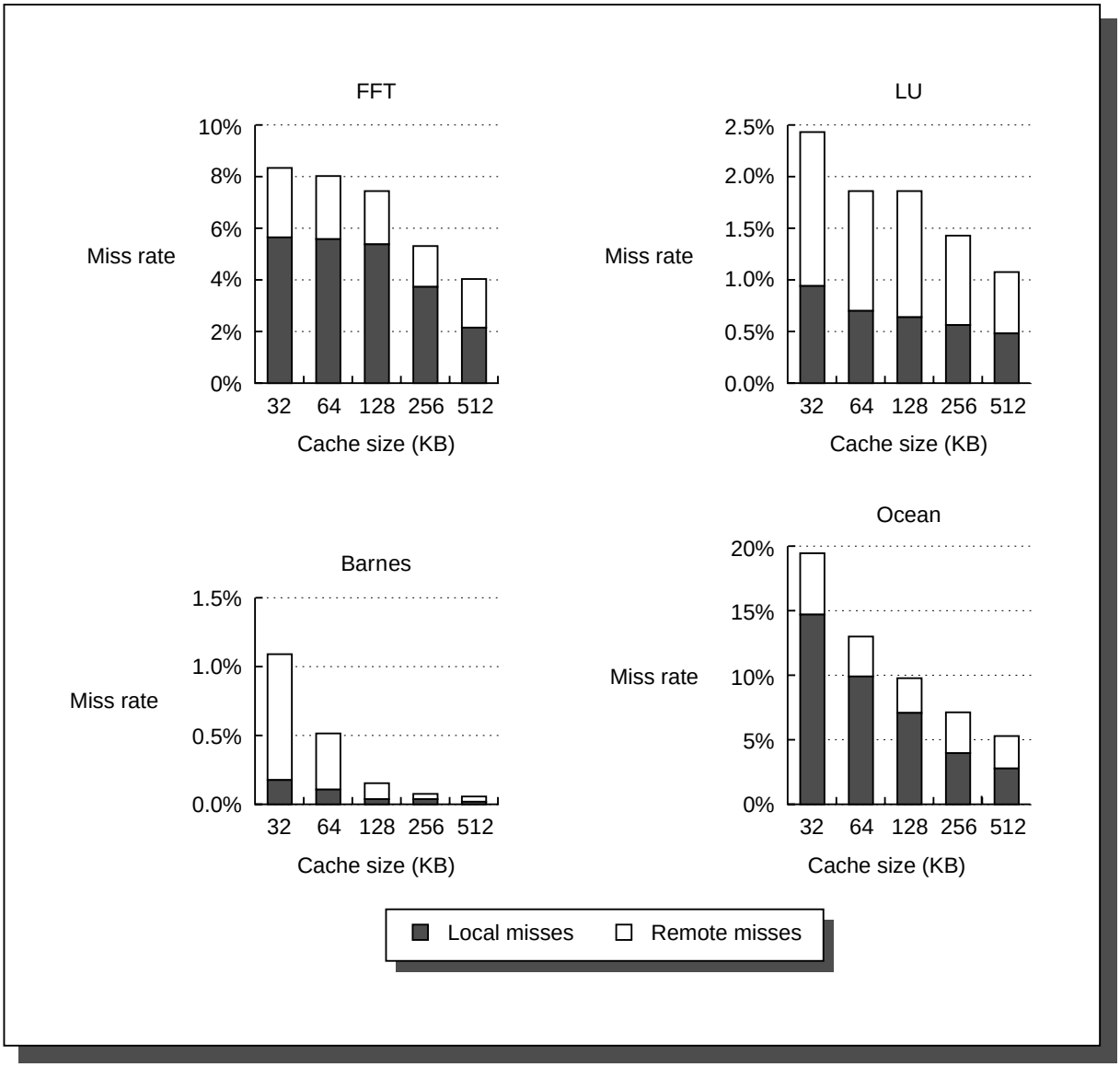


FIGURE 8.27 Miss rates decrease as cache sizes grow.

05-11

Directory Performance: Line Size

05-12

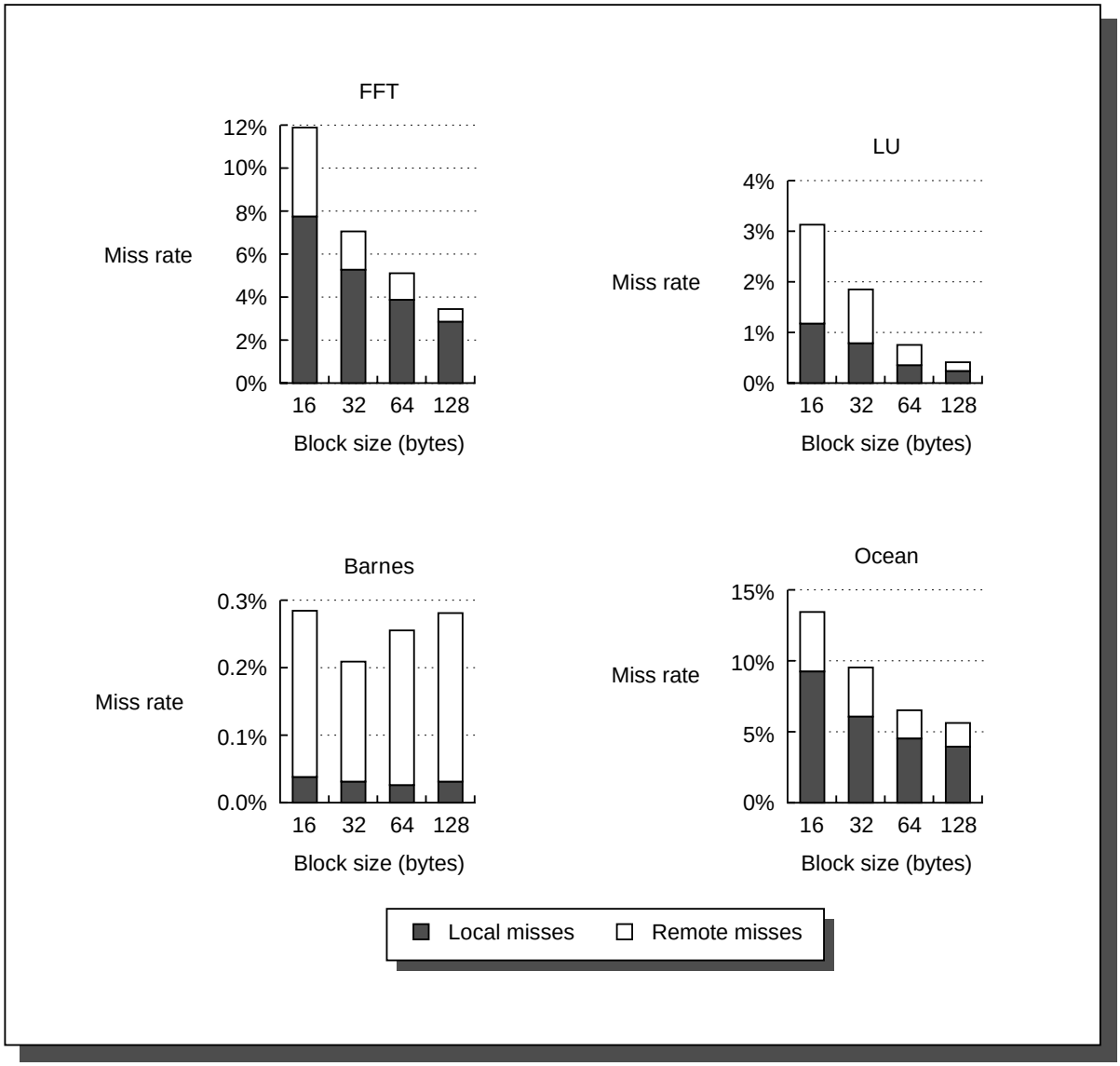


FIGURE 8.28 Data miss rate versus block size assuming a 128-KB cache and 64 processors in total.

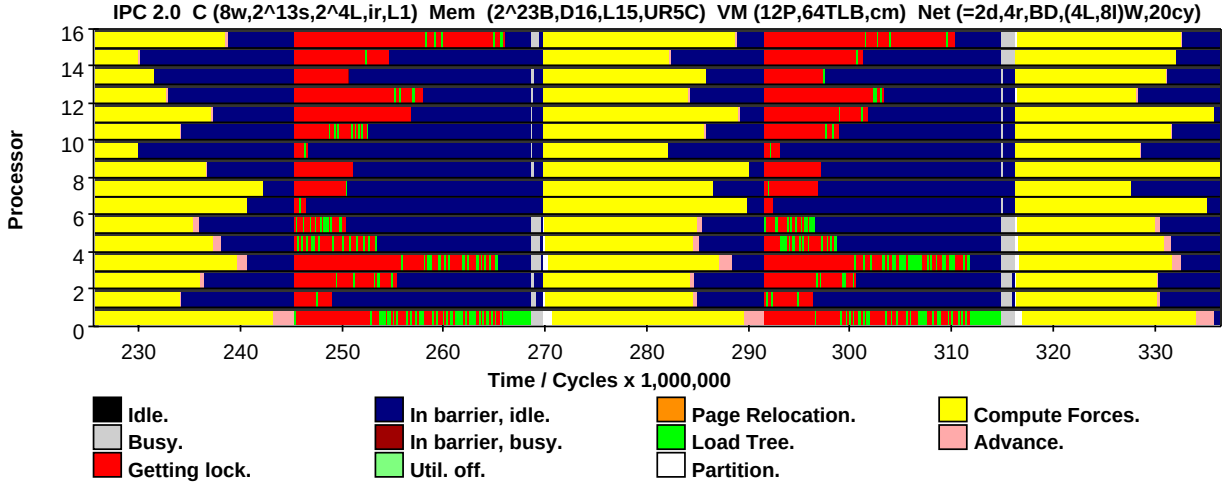
05-12

Directory Performance: Barnes

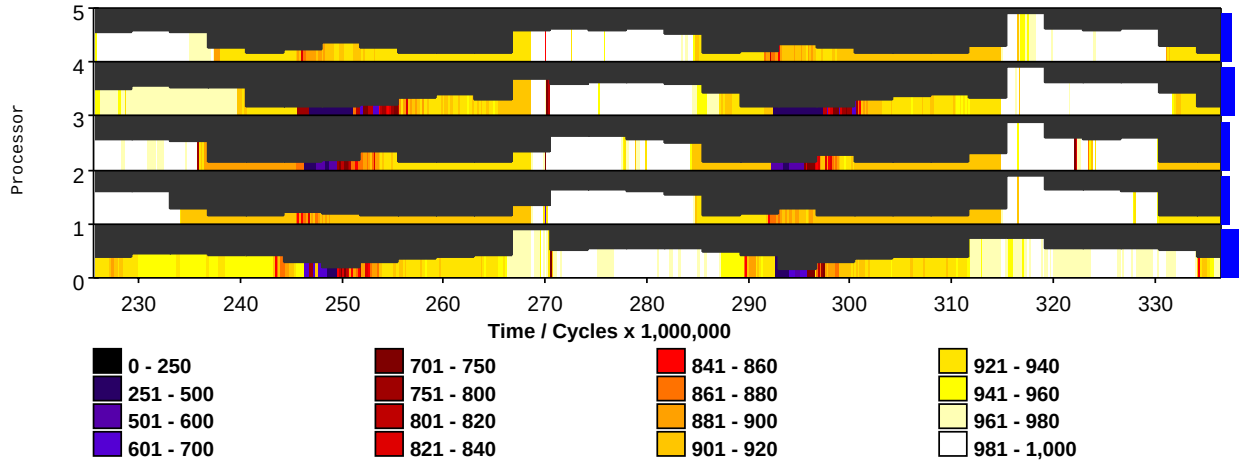
05-13

Processor States: Barnes, 16384 Particles

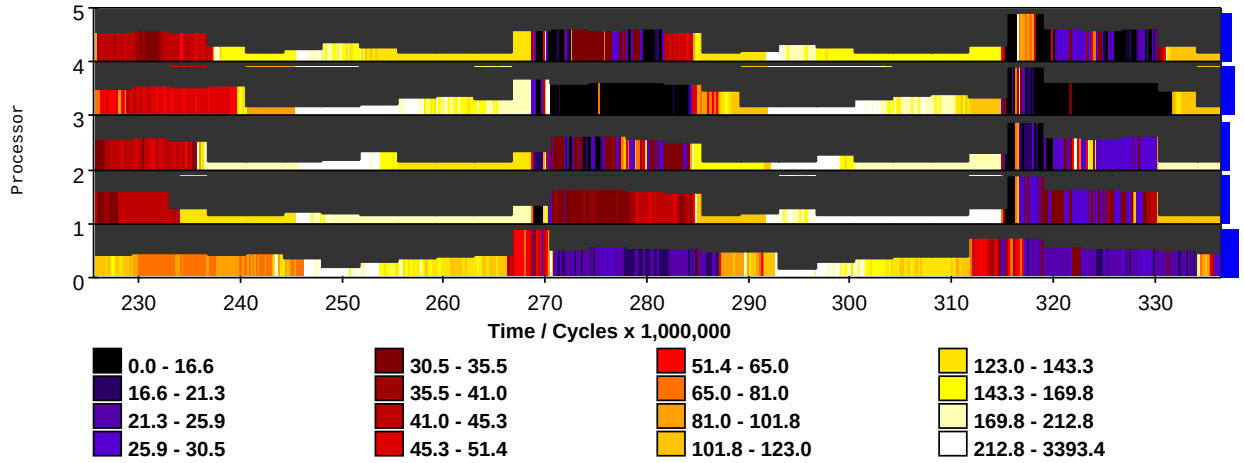
IPC 2.0 C (8w,2^13s,2^4L,ir,L1) Mem (2^23B,D16,L15,UR5C) VM (12P,64TLB,cm) Net (=2d,4r,BD,(4L,8I)W,20cy)



Cache Hit Ratio versus Time



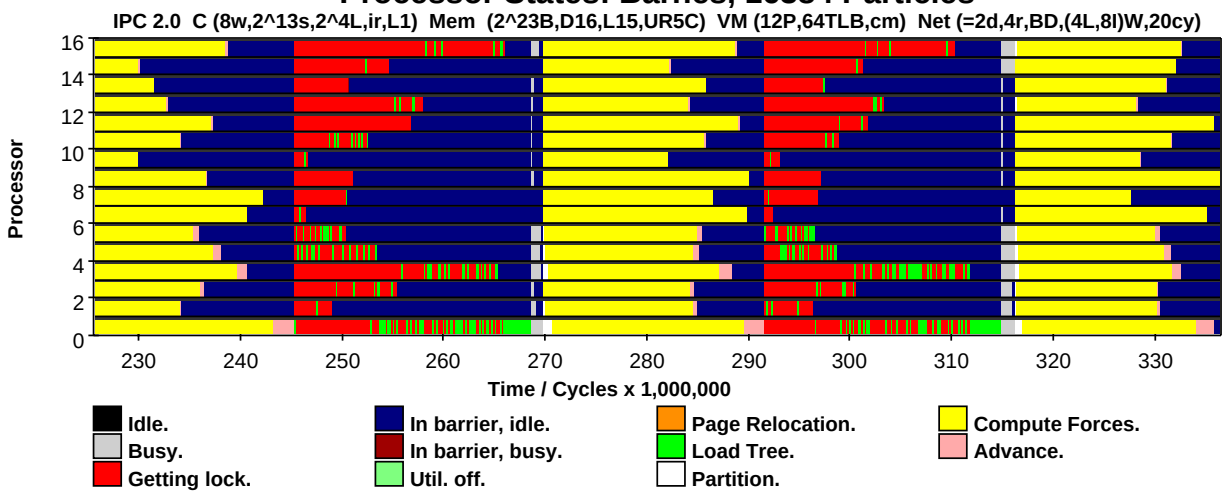
Memory Access Latency



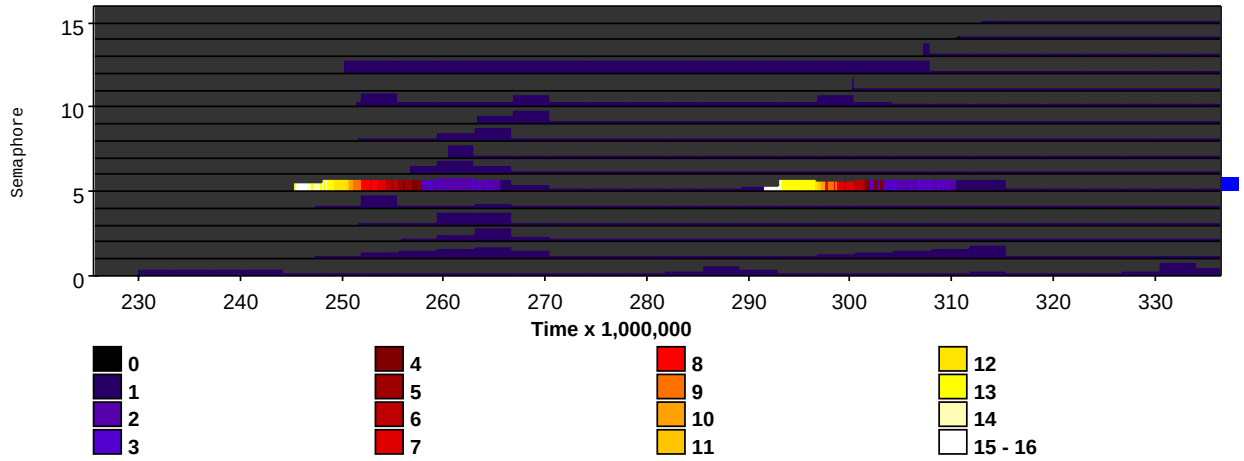
05-13

Directory Performance: Barnes

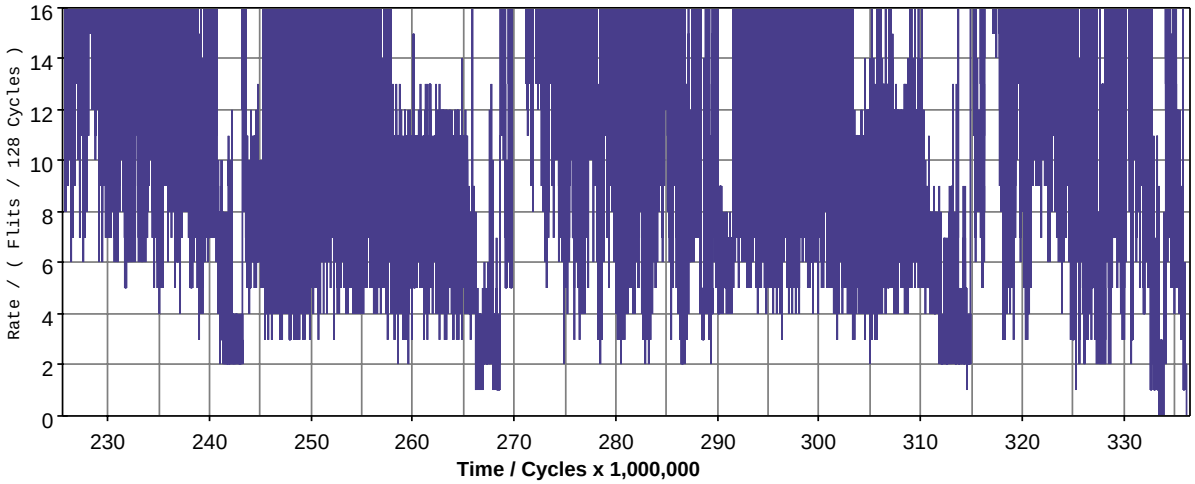
Processor States: Barnes, 16384 Particles



Threads Blocked on Semaphores



Network Traffic Volume (Arrival Rate)



Synchronization

05-15

Synchronization Instructions

Load and store instructions that can be used to synchronize processors.

Ordinary load and store instructions unsuitable.

Typical Instructions

- Atomic Exchange.
  `exch r1,0(r2)`
  Exchange r1 and contents of `0(r2)`.
- Fetch and Increment.
  Increment memory location, return old value.
- Instruction Pair: Load Linked (`ll`) and Store Conditional (`sc`).

  `ll r1,0(r2)`
  Load `r1` with contents of `0(r2)`.

  `sc r1,0(r2)`
  Store `r1` in `0(r2)` and set `r1` to 1 if last accessing instruction was `ll`.

Synchronization Instruction Implementation

05-16

Instructions implemented by protocol processor.

Atomic Exchange, Fetch and Increment

Exclusive copy obtained by cache, operation performed.

Load Linked (`ll`) and Store Conditional (`sc`).

Cache has *link register* holding address from last `ll` instruction.

Link register cleared if corresponding line invalidated ...
... or if processor interrupted.

When `sc` executed link register checked for address ...
... if found store completes.

Synchronization Instruction Example

05-17

Use of Load Linked and Store Conditional. (From text.)

Program performs atomic exchange of `r4` and contents of `0(r1)`.

```
try:
 add   r3, r4, r0  ! Move value to be stored to r3.
 ll    r2, 0(r1)   ! Load current value into r2.
 sc    r3, 0(r1)   ! Write r3 to 0(r1) and r3<-1 if store successful.
 beqz  r3, try     ! If r3 is zero, store failed so try again.
 add   r4, r2, r0  ! Move loaded value to r4.
```

Synchronization Operations

05-18

Used by high-level-language programmers.

Implemented using synchronization instructions.

Basic Operations

- Lock and Unlock.
  Provides exclusive access to resource.

  Also covered in Tera material.

- Barrier.
  Used when all tasks must finish one step before proceeding to next.

Synchronization Operation Examples

05-19

Lock / Unlock

```
/* Without lock some elements might be lost. */
lock(list);
new_element->next = list->head;
list->head = new_element;
unlock(list);
```

Barrier

```
/*  Shared: max, a, gmax.
    Local:  begin, end, i, myid. */

max[myid] = a[begin];
for(i=begin; i<end; i++) if( a[i] > max[myid] ) max[myid] = a[i];

barrier();

if( myid == 0 ) {
 gmax = max[0];
 for(i=1; i<nproc; i++) if( gmax < max[i] ) gmax = max[i];
}

barrier();

for(i=begin; i<end; i++) a[i] /= gmax;
```

Lock Implementations

05-20

Two Approaches

- Spin Lock
  Keep checking if lock available.

  Fast, but keeps processor busy.

- Queue
  Task requesting unavailable lock added to list.

  Uses less processor time and traffic under contention.

  Not covered in detail here.

Naïve Spin Lock Implementation

05-21

Very inefficient, especially with directory protocols.

Lock:

```
 ! When 0(r1) holds a 1, someone has lock.
 addi  r2, r0, #1   ! Set r2 to 1.
lockit:
 exch  r2, 0(r1)    ! Get old lock value.
 bnez  r2, lockit   ! If it's one, try again.
```

Unlock:

```
 add r2, r0, r0     ! Put unlock value in r2.
 sw  0(r1), r2      ! Write to lock.
```

Problem:

Execution of `exch` brings exclusive copy to cache.

Heavy protocol traffic even if two processors contending for lock.

That traffic might slow down or *starve* unlock operation.

Test and Exchange Spin Lock

05-22

Much more efficient.

Lock:

```
 addi r3, r0, #1   ! Prepare a "locked" value.
TEST:
 lw   r2, 0(r1)    ! Check lock state.
 bnez r2, TEST     ! If locked, check again.
 exch r2, 0(r1)    ! It was unlocked, try to lock it.
 bnez r2, TEST     ! Check if some other task got it first.
```

Unlock code same as spin lock.

Advantage:

Multiple processors can wait and not miss each iteration.

Problems:

When lock released, memory inundated with write requests.

Load Linked Spin Lock

05-23

Even more efficient.

Lock:

```
TEST:
 ll   r2, 0(r1)
 bnez r2, TEST     ! If locked, check again.
 addi r2, r0, #1   ! It's not locked! Prepare a "locked" value.
 sc   r2, 0(r1)    ! Try to write locked value.
 beqz r2, TEST     ! If store failed, try again.
```

Unlock code same as spin lock.

Advantage:

Multiple processors can wait and not miss each iteration.

Store conditional will not attempt to get exclusive copy if another processor invalidates line.

Problems:

When lock released, memory is inundated with write requests, though all but one abandoned.

Barrier Implementation

05-24

Operation

First caller sets release condition to false.

All callers increment an arrival-at-barrier count.

Last caller sets release condition to true.

Not suitable for loops.

Code

```
// total: number of tasks.
// arrival_count: number of callers so far.
// release: true when it's okay to leave barrier.
lock(arrival_count);
if( arrival_count == 0 ) release = false;
arrival_count++;
unlock(arrival_count);
if( arrival_count == total ) {
 arrival_count = 0;
 release = true;
} else {
 spin(release);
}
```

Alternate Barrier Implementation

05-25

Operation

Update value used to indicate true. (Changes each call.)

Increment arrival count and get old value.

If last in, reset counter and set release to true, otherwise wait.

Code

```
// total: number of tasks.
// arrival_count: number of callers so far.
// release: equal to true_value when it's okay to leave barrier.
true_value = ! true_value;
myplace = fetch_and_increment(arrival_count);
if( myplace == total - 1 )
 arrival_count = 0;
 release = true_value;
} else {
 spin( release == true_value );
}
```

Memory Consistency Models

05-26

Consistency v. Coherence

Coherence: All processors "see" *consistent* view of memory.

Coherence: Constrains reads and writes to a *particular* address.

Consistency: Specifies when processors "see" values.

Consistency: Constrains reads and writes to any address.

Consistency Models

Many models proposed and in use.

Tradeoff is ease of programming v. speed of implementation.

Many programs easy to write regardless of model.

Consistency Model, Slowest to Fastest

- Sequential Consistency
- Processor Consistency (Total Store Ordering)
- Partial Store Ordering
- Weak Ordering
- Release Consistency

Sequential Consistency

05-27

Defined in terms of (very impractical) abstract machine in which:

processors execute instructions one at a time (no pipelining, etc.),

only one processor at a time is allowed to execute

once a processor starts an instruction it is allowed to complete it without interruption

there are no caches.

A system is sequentially consistent if ...

... the values returned by the loads during any execution of any program ...

... can also be returned when the program is run on the abstract machine.

Sequential Consistency Implementation

05-28

Implementation can be much less restrictive than abstract machine.

Implementation

Use protocol that waits for invalidation acknowledgments.

Processor does not start a memory operation until previous one completes.

Disadvantage: writes slow down execution.

Relaxed Models

05-29

Sequential consistency most restrictive, so other models called *relaxed*.

Synchronization operations needed with relaxed models . . .
. . . to impose ordering lost in model.

E.g., to ensure a read is executed after a write.

Synchronized Program
A program in which for all shared data synchronization operations are executed after the data is written and before that data is read.

Release (or equivalent) synchronization operation used after write.
Acquire (or equivalent) synchronization operation used before read.

Example:

```
// a shared.
// x, y, w are local.
a[x]=y; // Write data
release();
...
acquire();
z = a[w]; // Read data (written at other processor)
```

Same code executing on multiple processors.

If acquire executed before release written data will be read.

Description of Relaxed Models

05-30

Defined in terms of access ordering that is enforced.

E.g., Read after write.

Notation

$X \rightarrow Y$ where $X, Y \in \{R, W, S, S_A, S_R\}$

If X appears before Y in program order *at a processor* then ...
... the memory system guarantees that X will complete before Y.

Note that X and Y addresses not necessarily same.

Relaxed Model Definitions

All require coherent memory systems.

Enforced orderings specified in definition.

Definition may also specify *synchronization ordering*:

$S \rightarrow W$, $S \rightarrow R$, $S \rightarrow S$, $R \rightarrow S$, and $W \rightarrow S$.

Operation Completion

Write: last invalidate reaches destination. (Practically, writing cache gets confirmation that all acknowledgments received.)

Read: time value moved from cache to processor.

Operations must *appear* to complete in order, even though they may complete out of order.

Definition of Sequential Consistency

Enforces: synchronization ordering and $R \rightarrow R$, $R \rightarrow W$, $W \rightarrow W$, and $W \rightarrow R$.

Processor Consistency Model

05-31

Processor Consistency (Also called Total Store Ordering)

Enforces synchronization ordering and $R \rightarrow R$, $R \rightarrow W$, and $W \rightarrow W$.

(Does not enforce $W \rightarrow R$.)

Advantage: Reads do not have to wait for writes.

Processor Consistency Implementation

Queue writes, execute in order.

Reads and writes wait for any preceding read to finish.

Hardware requirements

Protocol waits for invalidation acknowledgments.

Protocol processor must be able to simultaneously handle a read and write miss.

Partial Store Ordering Model

05-32

Partial Store Ordering

Enforces synchronization ordering and $R \rightarrow R$ and $R \rightarrow W$

(Does not enforce $W \rightarrow R$ and $W \rightarrow W$.)

Advantage: writes can be pipelined.

Implementation

Writes and reads wait for any preceding reads to complete.

Hardware requirements

Protocol processor must be able to simultaneously handle a single read and multiple write misses.

05-33

Weak Ordering

Enforces synchronization ordering.

(Does not enforce $R \rightarrow R$, $R \rightarrow W$, $W \rightarrow W$, and $W \rightarrow R$.)

Release Consistency

Enforces $S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, and $S_R \rightarrow S_R$.

Advantage: acquire and release slightly faster than ordinary synchronization operation.

Execution Under Memory Models Example

05-34

Dekker's Algorithm. Requires sequential consistency.

Code on next slide.

Code runs on two processors as indicated in comments.

Two variables, `x` and `y` should be changed atomically ...
... that is either both changed or neither changed.

Dekker's Algorithm, Sequential Consistency Implementation

05-35

```
/* Initially, x=0 and y=0.
   Variables x and y only modified in critical regions.
*/

/* Processor 1 */

/* Obtain Lock */
do{
 a = 0;
 random_short_wait();
 a = 1;
} while ( b == 1 );
/* Lock obtained, critical region start. */
x = 1;
y = 2;
/* Critical region end. */
a = 0;  /* Release Lock. */


/* Processor 2 */

/* Obtain Lock */
do{
 b = 0;
 random_short_wait();
 b = 1;
} while ( a == 1 );
/* Lock obtained, critical region start. */
if( x==0 && y==0 ) {
 /* Processor 1 (that's us) in in critical region first. */
} else if( x==1 && y==2 ) {
 /* Processor 2 in critical region first. */
} else {
 /* Execution should not reach this point. */
}
/* Critical region end. */
b = 0;  /* Release Lock. */
```

Execution of Dekker's Algorithm

05-36

Execution of Sequential Consistency Implementation

On a sequentially consistent model runs correctly.

Does not work on processor consistency model.

Problem under processor consistency model.

To obtain lock processor 1 executes:

```
do{
 a = 0;  /* W11 */
 random_short_wait();
 a = 1;  /* W12 */
} while ( b == 1 ); /* R1 */
```

To obtain lock processor 2 executes:

```
do{
 b = 0;  /* W21 */
 random_short_wait();
 b = 1;  /* W22 */
} while ( a == 1 ); /* R2 */
```

Under processor consistency read doesn't have to wait for write.

Possible order (position indicates completion times):

```
W11  R1   W12
W21  R2   W22
```

Since both reads execute before 1 written, both will enter critical region.

```
/* Initially, x=0 and y=0.
   Variables x and y only modified in critical regions.
*/

/* Processor 1 */
/* Obtain Lock */
do{
 a = 0;    /* W11 */
 random_short_wait();
 a = 1;    /* W12 */
 synch();  /* S1 */
} while ( b == 1 );  /* R1 */
/* Lock obtained, critical region start. */
x = 1;
y = 2;
/* Critical region end. */
a = 0;  /* Release Lock. */

/* Processor 2 */
/* Obtain Lock */
do{
 b = 0;    /* W21 */
 random_short_wait();
 b = 1;    /* W22 */
 synch();  /* S2 */
} while ( a == 1 );  /* R2 */
/* Lock obtained, critical region start. */
if( x==0 && y==0 ) {
 /* Processor 1 (that's us) in in critical region first. */
} else if( x==1 && y==2 ) {
 /* Processor 2 in critical region first. */
} else {
 /* Execution should not reach this point. */
}
/* Critical region end. */
b = 0;  /* Release Lock. */
```

Execution of Processor Consistency Implementation

05-38

Added `synch()` forces read to wait for preceding write.

Execution:

```
W11       W12      S1    R1
W21       W22      S2    R2
```

Executes correctly under processor consistency and sequential consistency.

Does not work under partial store order.

Problem: Because writes not ordered processor 2 may "see" unlock (`a=0`) and `x=1` but may not see `y=2`.

Solution is to place a `synch()` before `a=0`.