Shared Memory Multiprocessors

Logical design and software interactions

Shared Memory Multiprocessors

Symmetric Multiprocessors (SMPs)

- Symmetric access to all of main memory from any processor

Dominate the server market

- Building blocks for larger systems; arriving to desktop

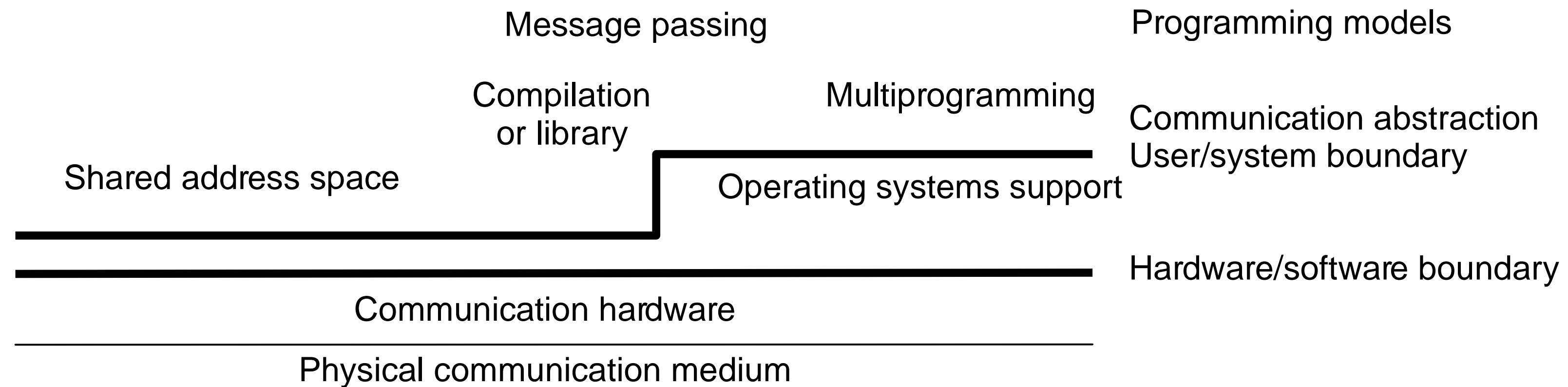
Attractive as throughput servers and for parallel programs

- Fine-grain resource sharing
- Uniform access via loads/stores
- Automatic data movement and coherent replication in caches
- Useful for operating system too

Normal uniprocessor mechanisms to access data (reads and writes)

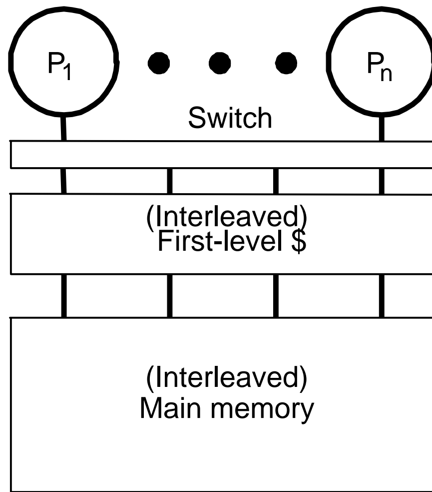
- Key is extension of memory hierarchy to support multiple processors

Supporting Programming Models

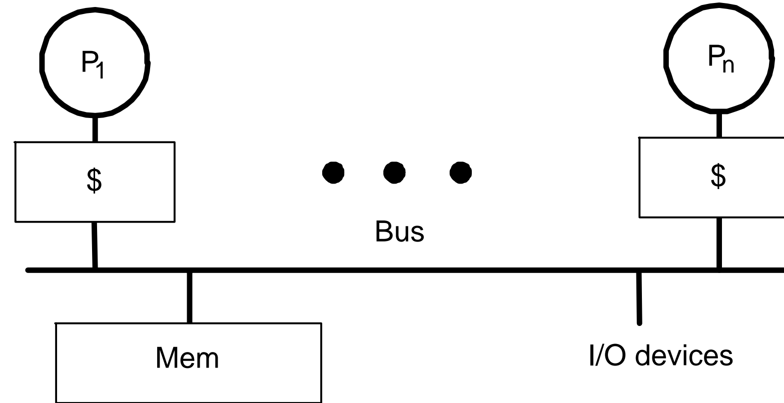


- Address translation and protection in hardware (hardware SAS)
- Message passing using shared memory buffers
  - can be very high performance since no OS involvement necessary
- Focus here on supporting coherent shared address space

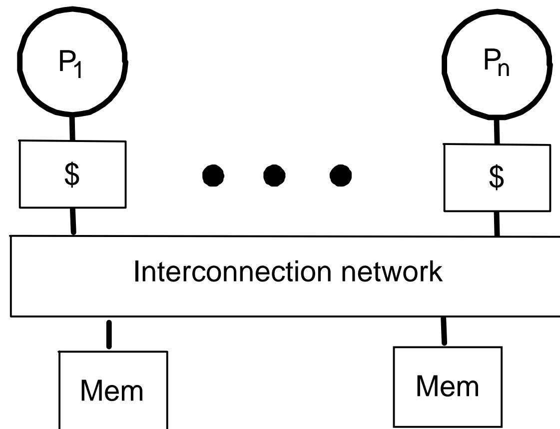
Natural Extensions of Memory System



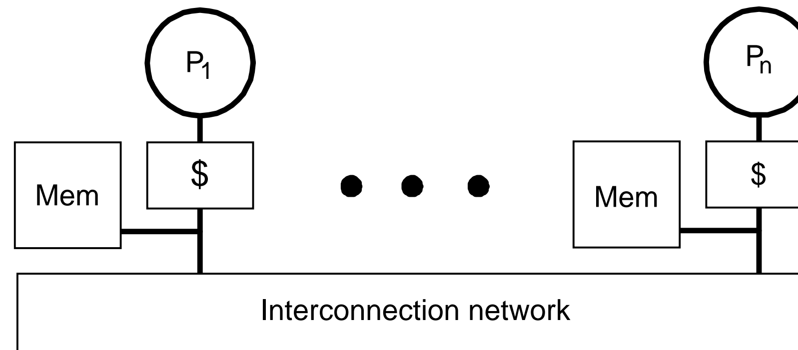
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Caches and Cache Coherence

Caches play key role in all cases

- Reduce average data access time
- Reduce bandwidth demands placed on shared interconnect

But private processor caches create a problem

- Copies of a variable can be present in multiple caches
- A write by one processor may not become visible to others
  - They'll keep accessing stale value in their caches
- *Cache coherence* problem
- Need to take actions to ensure visibility

Focus: Bus-based, Centralized Memory

Shared cache

- Low-latency sharing and prefetching across processors
- Sharing of working sets
- No coherence problem (and hence no false sharing either)
- But high bandwidth needs and negative interference (e.g. conflicts)
- Hit and miss latency increased due to intervening switch and cache size
- Mid 80s: to connect couple of processors on a board (Encore, Sequent)
- Today: for multiprocessor on a chip (for small-scale systems or nodes)

Dancehall

- No longer popular: everything is uniformly *far* away

Distributed memory

- Most popular way to build scalable systems, discussed later

Outline

Coherence and Consistency

Snooping Cache Coherence Protocols

Quantitative Evaluation of Cache Coherence Protocols

Synchronization

Implications for Parallel Software

A Coherent Memory System: Intuition

Reading a location should return latest value written (by any process)

Easy in uniprocessors

- Except for I/O: coherence between I/O devices and processors
- But infrequent so software solutions work
  - uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches

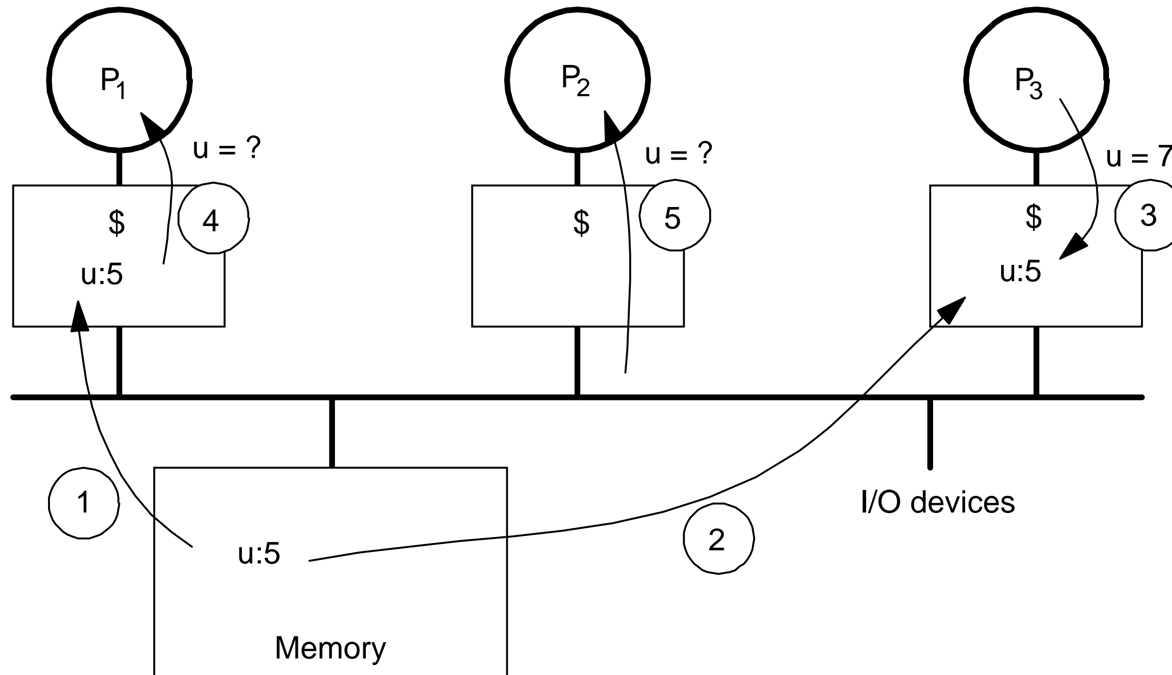
Would like same to hold when processes run on different processors

- E.g. as if the processes were interleaved on a uniprocessor

But coherence problem much more critical in multiprocessors

- Pervasive
- Performance-critical
- Must be treated as a basic hardware design issue

Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

Problems with the Intuition

Recall: Value returned by read should be last value written

But “last” is not well-defined

Even in seq. case, last defined in terms of program order, not time

- Order of operations in the machine language presented to processor
- “Subsequent” defined in analogous way, and well defined

In parallel case, program order defined within a process, but need to make sense of orders across processes

Must define a meaningful semantics

Some Basic Definitions

Extend from definitions in uniprocessors to those in multiprocessors

Memory operation: a single read (load), write (store) or read-modify-write access to a memory location

- Assumed to execute atomically w.r.t each other

Issue: a memory operation issues when it leaves processor's internal environment and is presented to memory system (cache, buffer …)

Perform: operation appears to have taken place, as far as processor can tell from other memory operations it issues

- A write performs w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write
- A read perform w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read

In multiprocessors, stay same but replace "the" by "a" processor

- Also, *complete*: perform with respect to all processors
- Still need to make sense of order in operations from different processes

Sharpening the Intuition

Imagine a single shared memory and no caches

- Every read and write to a location accesses the same physical location
- Operation completes when it does so

Memory imposes a *serial* or *total order* on operations to the location

- Operations to the location from a given processor are in program order
- The order of operations to the location from different processors is some interleaving that preserves the individual program orders

"Last" now means most recent in a hypothetical serial order that maintains these properties

For the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look, i.e. to read)

Note that the total order is never really constructed in real systems

- Don't even want memory, or any hardware, to see all operations

But program should behave as if some serial order is enforced

- Order in which things appear to happen, not actually happen

Formal Definition of Coherence

Results of a program: values returned by its read operations

A memory system is *coherent* if the results of any execution of a program are such that each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

1. operations issued by any particular process occur in the order issued by that process, and

2. the value returned by a read is the value written by the last write to that location in the serial order

Two necessary features:

- *Write propagation*: value written must become visible to others
- *Write serialization*: writes to location seen in same order by all
  - if I see w1 after w2, you should not see w2 before w1
  - no need for analogous read serialization since reads not visible to others

Cache Coherence Using a Bus

Built on top of two fundamentals of uniprocessor systems

- Bus transactions
- State transition diagram in cache

Uniprocessor bus transaction:

- Three phases: arbitration, command/address, data transfer
- All devices observe addresses, one is responsible

Uniprocessor cache states:

- Effectively, every block is a finite state machine
- Write-through, write no-allocate has two states: valid, invalid
- Writeback caches have one more state: modified (“dirty”)

Multiprocessors extend both these somewhat to implement coherence

Snooping-based Coherence

Basic Idea

Transactions on bus are visible to all processors

Processors or their representatives can snoop (monitor) bus and take action on relevant events (e.g. change state)

Implementing a Protocol

Cache controller now receives inputs from both sides:

- Requests from processor, bus requests/responses from snooper

In either case, takes zero or more actions

- Updates state, responds with data, generates new bus transactions

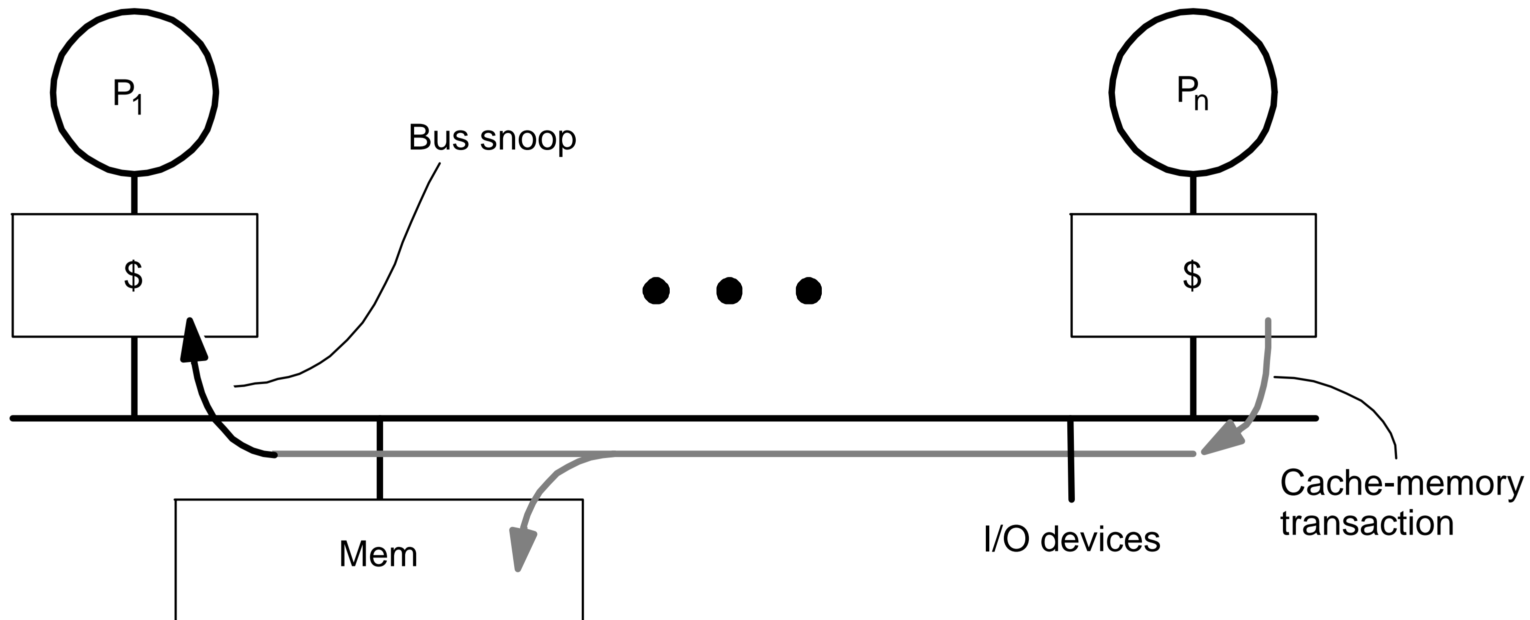
Protocol is distributed algorithm: cooperating state machines

- Set of states, state transition diagram, actions

Granularity of coherence is typically cache block

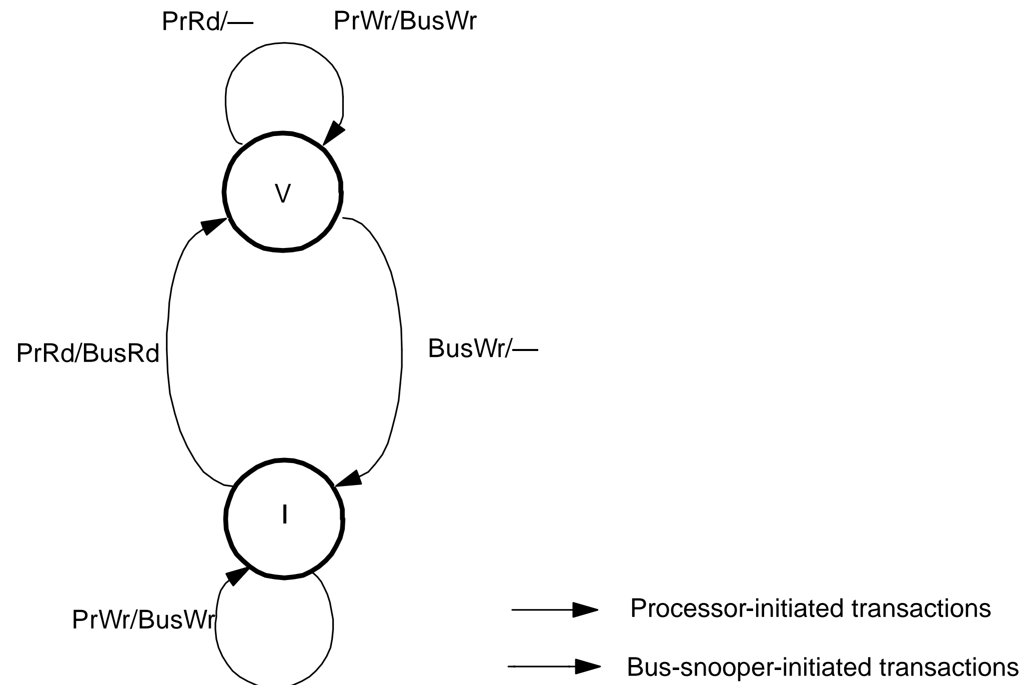
- Like that of allocation in cache and transfer to/from cache

Coherence with Write-through Caches



- Key extensions to uniprocessor: snooping, invalidating/updating caches
  - no new states or bus transactions in this case
  - invalidation- versus update-based protocols
- Write propagation: even in inval case, later reads will see new value
  - inval causes miss on later access, and memory up-to-date via write-through

Write-through State Transition Diagram



- Two states per block in each cache, as in uniprocessor
  - state of a block can be seen as *p*-vector
- Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Write will invalidate all other caches (no local change of state)
  - can have multiple simultaneous readers of block,but write invalidates them

Is it Coherent?

Construct total order that satisfies program order, write serialization?

Assume atomic bus transactions and memory operations for now

- all phases of one bus transaction complete before next one starts
- processor waits for memory operation to complete before issuing next
- with one-level cache, assume invalidations applied during bus xaction
- (we'll relax these assumptions in more complex systems later)

All writes go to bus + atomicity

- Writes serialized by order in which they appear on bus (*bus order*)
- Per above assumptions, invalidations applied to caches in bus order

How to insert reads in this order?

- Important since processors see writes through reads, so determines whether write serialization is satisfied
- But read hits may happen independently and do not appear on bus or enter directly in bus order

Ordering Reads

Read misses: appear on bus, and will see last write in bus order

Read hits: do not appear on bus

- But value read was placed in cache by either
  - most recent write by this processor, or
  - most recent read miss by this processor
- Both these transactions appear on the bus
- So reads hits also see values as being produced in consistent bus order

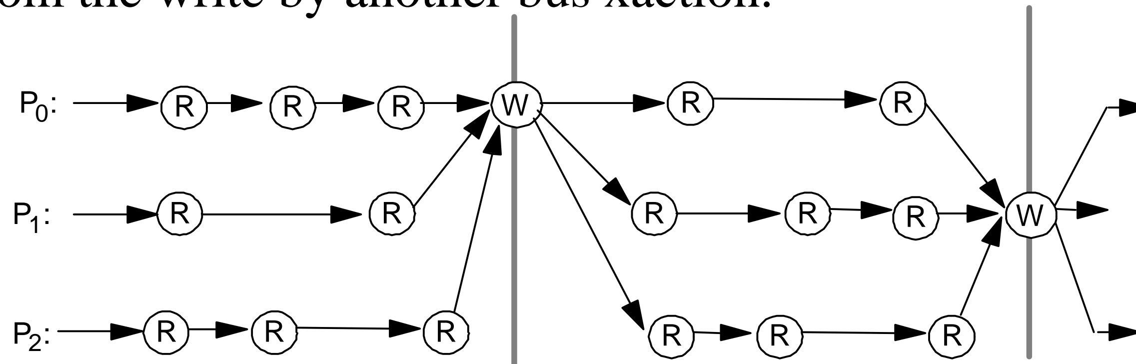
Determining Orders More Generally

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

Read is subsequent to write W if read generates bus xaction that follows that for W.

Write is subsequent to read or write M if M generates bus xaction and the xaction for the write follows that for M.

Write is subsequent to read if read does not generate a bus xaction and is not already separated from the write by another bus 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

Problem with Write-Through

High bandwidth requirements

- Every write from every processor goes to shared bus and memory
- Consider 200MHz, 1CPI processor, and 15% instrs. are 8-byte stores
- Each processor generates 30M stores or 240MB data per second
- 1GB/s bus can support only about 4 processors without saturating
- Write-through especially unpopular for SMPs

Write-back caches absorb most writes as cache hits

- Write hits don't go on bus
- But now how do we ensure write propagation and serialization?
- Need more sophisticated protocols: large design space

But first, let's understand other ordering issues

Memory Consistency

Writes to a location become visible to all in the same order

But when does a write become visible

- How to establish orders between a write and a read by different procs?
  - Typically use event synchronization, by using more than one location

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

- Intuition not guaranteed by coherence
- Sometimes expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence doesn’t help: pertains only to single location

Another Example of Orders

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's the intuition?
- Whatever it is, we need an ordering model for clear semantics
  - across different locations as well
  - so programmers can reason about what results are possible
- This is the memory consistency model

Memory Consistency Model

Specifies constraints on the order in which memory operations (from any process) can *appear to execute* with respect to one another

- What orders are preserved?
- Given a load, constrains the possible values returned by it

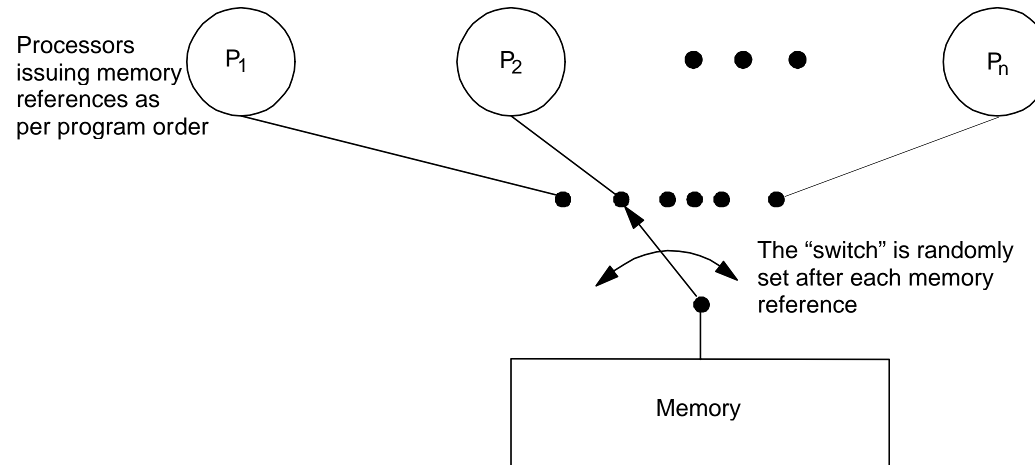
Without it, can't tell much about an SAS program's execution

Implications for both programmer and system designer

- Programmer uses to reason about correctness and possible results
- System designer can use to constrain how much accesses can be reordered by compiler or hardware

Contract between programmer and system

Sequential Consistency



- (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
- Programmer's intuition is maintained

"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]

What Really is Program Order?

Intuitively, order in which operations appear in source code

- Straightforward translation of source code to assembly
- At most one memory operation per instruction

But not the same as order presented to hardware by compiler

So which is program order?

Depends on which layer, and who's doing the reasoning

We assume order as seen by programmer

SC Example

What matters is order in which *appears to execute*, not *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)
- we know 1a->1b and 2a->2b by program order
- 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 program order
  - appears just like 1a->1b->2a->2b as visible from results
- actual execution 1b->2a->2b-> is not SC

Implementing SC

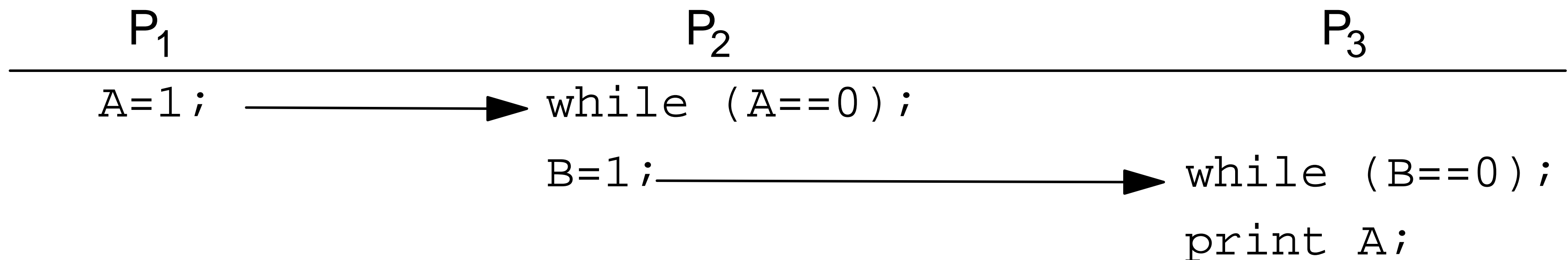
Two kinds of requirements

- Program order
  - memory operations issued by a process must appear to become visible (to others and itself) in program order
- Atomicity
  - in the overall total order, one memory operation should appear to complete with respect to all processes before the next one is issued
  - needed to guarantee that total order is consistent across processes
  - tricky part is making writes atomic

Write Atomicity

Write Atomicity: Position in total order at which a write appears to perform should be the same for all processes

- Nothing a process does after it has seen the new value produced by a write W should be visible to other processes until they too have seen W
- In effect, extends write serialization to writes from multiple processes



- Transitivity implies A should print as 1 under SC
- Problem if P_2 leaves loop, writes B, and P_3 sees new B but old A (from its cache, say)

More Formally

Each process's program order imposes partial order on set of all operations

Interleaving of these partial orders defines a total order on all operations

Many total orders may be SC (SC does not define particular interleaving)

SC Execution: An execution of a program is SC if the results it produces are the same as those produced by some possible total order (interleaving)

SC System: A system is SC if any possible execution on that system is an SC execution

Sufficient Conditions for SC

- Every process issues memory operations in program order
- After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
- After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity)

Sufficient, not necessary, conditions

Clearly, compilers should not reorder for SC, but they do!

- Loop transformations, register allocation (eliminates!)

Even if issued in order, hardware may violate for better performance

- Write buffers, out of order execution

Reason: uniprocessors care only about dependences to same location

- Makes the sufficient conditions very restrictive for performance

Our Treatment of Ordering

Assume for now that compiler does not reorder

Hardware needs mechanisms to detect:

- Detect write completion (read completion is easy)
- Ensure write atomicity

For all protocols and implementations, we will see

- How they satisfy coherence, particularly write serialization
- How they satisfy sufficient conditions for SC (write completion and write atomicity)
- How they can ensure SC but not through sufficient conditions

Will see that centralized bus interconnect makes it easier

SC in Write-through Example

Provides SC, not just coherence

Extend arguments used for coherence

- Writes and read misses to *all locations* serialized by bus into bus order
- If read obtains value of write W, W guaranteed to have completed
  - since it caused a bus transaction
- When write W is performed w.r.t. any processor, all previous writes in bus order have completed

Design Space for Snooping Protocols

No need to change processor, main memory, cache …

- Extend cache controller and exploit bus (provides serialization)

Focus on protocols for write-back caches

Dirty state now also indicates exclusive ownership

- Exclusive: only cache with a valid copy (main memory may be too)
- Owner: responsible for supplying block upon a request for it

Design space

- Invalidation versus Update-based protocols
- Set of states

Invalidation-based Protocols

Exclusive means can modify without notifying anyone else

- i.e. without bus transaction
- Must first get block in exclusive state before writing into it
- Even if already in valid state, need transaction, so called a write miss

Store to non-dirty data generates a *read-exclusive* bus transaction

- Tells others about impending write, obtains exclusive ownership
  - makes the write visible, i.e. write is performed
  - may be actually observed (by a read miss) only later
  - write hit made visible (performed) when block updated in writer's cache
- Only one RdX can succeed at a time for a block: serialized by bus

Read and Read-exclusive bus transactions drive coherence actions

- Writeback transactions also, but not caused by memory operation and quite incidental to coherence protocol
  - note: replaced block that is not in modified state can be dropped

Update-based Protocols

A write operation updates values in other caches

- New, update bus transaction

Advantages

- Other processors don't miss on next access: reduced latency
  - In invalidation protocols, they would miss and cause more transactions
- Single bus transaction to update several caches can save bandwidth
  - Also, only the word written is transferred, not whole block

Disadvantages

- Multiple writes by same processor cause multiple update transactions
  - In invalidation, first write gets exclusive ownership, others local

Detailed tradeoffs more complex

Invalidate versus Update

Basic question of program behavior

- Is a block written by one processor read by others before it is rewritten?

Invalidation:

- Yes => readers will take a miss
- No => multiple writes without additional traffic
  - and clears out copies that won't be used again

Update:

- Yes => readers will not miss if they had a copy previously
  - single bus transaction to update all copies
- No => multiple useless updates, even to dead copies

Need to look at program behavior and hardware complexity

Invalidation protocols much more popular (more later)

- Some systems provide both, or even hybrid

Basic MSI Writeback Inval Protocol

States

- Invalid (I)
- Shared (S): one or more
- Dirty or Modified (M): one only

Processor Events:

- PrRd (read)
- PrWr (write)

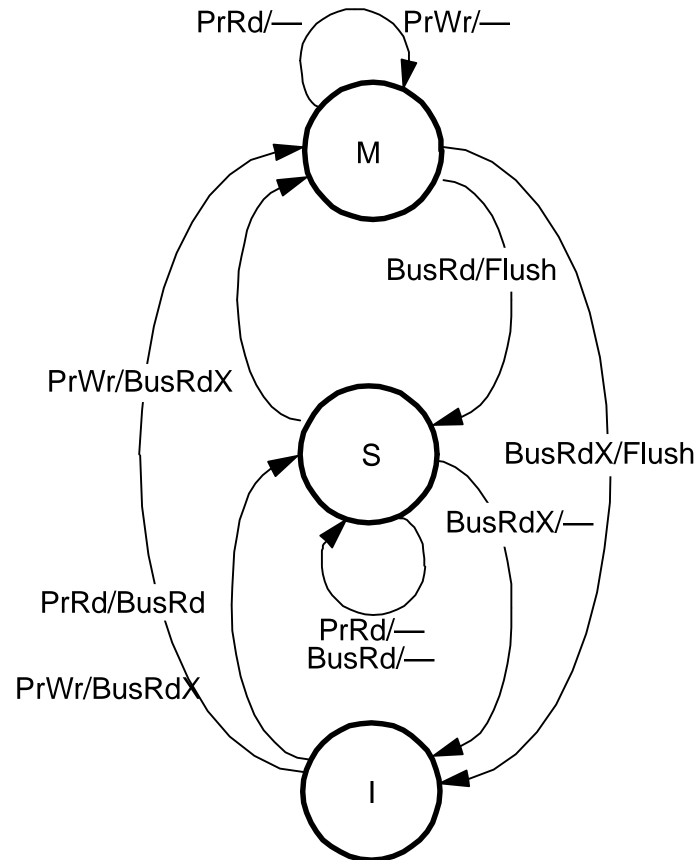
Bus Transactions

- BusRd: asks for copy with no intent to modify
- BusRdX: asks for copy with intent to modify
- BusWB: updates memory

Actions

- Update state, perform bus transaction, flush value onto bus

State Transition Diagram



- Write to shared block:
  - Already have latest data; can use upgrade (BusUpgr) instead of BusRdX
- Replacement changes state of two blocks: outgoing and incoming

Satisfying Coherence

Write propagation is clear

Write serialization?

- All writes that appear on the bus (BusRdX) ordered by the bus
  - Write performed in writer’s cache before it handles other transactions, so ordered in same way even w.r.t. writer
- Reads that appear on the bus ordered wrt these
- Write that don’t appear on the bus:
  - sequence of such writes between two bus xactions for the block must come from same processor, say P
  - in serialization, the sequence appears between these two bus xactions
  - reads by P will seem them in this order w.r.t. other bus transactions
  - reads by other processors separated from sequence by a bus xaction, which places them in the serialized order w.r.t the writes
  - so reads by all processors see writes in same order

Satisfying Sequential Consistency

1. Appeal to definition:
   - Bus imposes total order on bus xactions for all locations
   - Between xactions, procs perform reads/writes locally in program order
   - So any execution defines a natural partial order
     - M_j subsequent to M_i if (I) follows in program order on same processor, (ii) M_j generates bus xaction that follows the memory operation for M_i
   - In segment between two bus transactions, any interleaving of ops from different processors leads to consistent total order
   - In such a segment, writes observed by processor P serialized as follows
     - Writes from other processors by the previous bus xaction P issued
     - Writes from P by program order
2. Show sufficient conditions are satisfied
   - Write completion: can detect when write appears on bus
   - Write atomicity: if a read returns the value of a write, that write has already become visible to all others already (can reason different cases)

Lower-level Protocol Choices

BusRd observed in M state: what transitition to make?

Depends on expectations of access patterns

- S: assumption that I'll read again soon, rather than other will write
  - good for mostly read data
  - what about "migratory" data
    - I read and write, then you read and write, then X reads and writes...
    - better to go to I state, so I don't have to be invalidated on your write
- Synapse transitioned to I state
- Sequent Symmetry and MIT Alewife use adaptive protocols

Choices can affect performance of memory system (later)

MESI (4-state) Invalidation Protocol

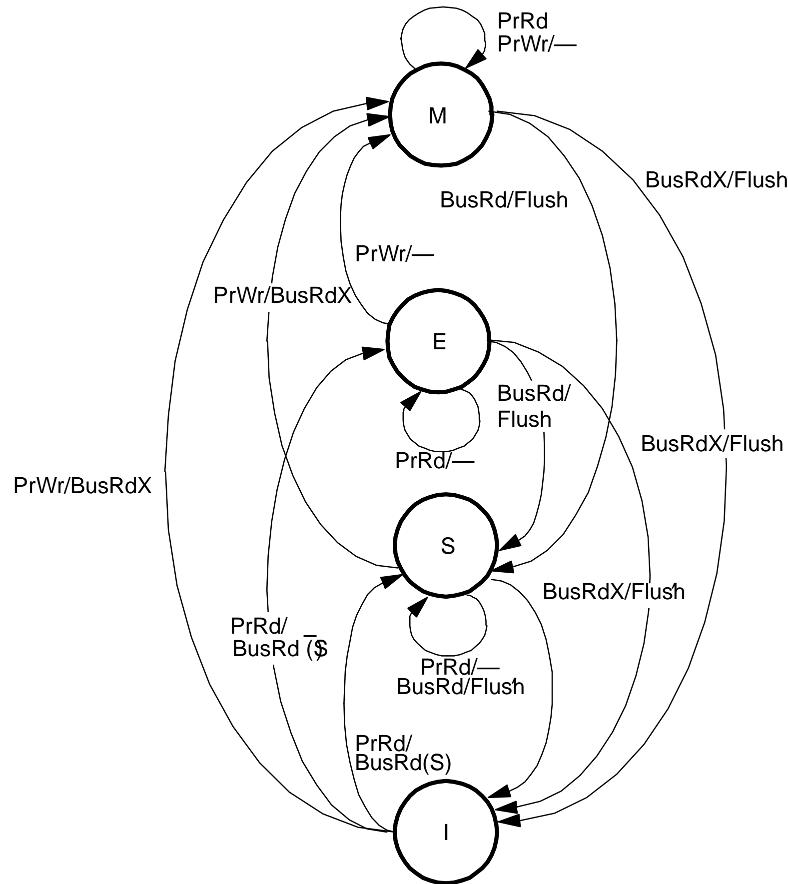
Problem with MSI protocol

- Reading and modifying data is 2 bus xactions, even if noone sharing
  - e.g. even in sequential program
  - BusRd (I->S) followed by BusRdX or BusUpgr (S->M)

Add *exclusive* state: write locally without xaction, but not modified

- Main memory is up to date, so cache not necessarily owner
- States
  - invalid
  - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I -> E on PrRd if noone else has copy
  - needs “shared” signal on bus: wired-or line asserted in response to BusRd

MESI State Transition Diagram



- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache sharing (see next), only one cache flushes data
- MOESI protocol: Owned state: exclusive but memory not valid

Lower-level Protocol Choices

Who supplies data on miss when not in M state: memory or cache

Original, *lllinois* MESI: cache, since assumed faster than memory

- *Cache-to-cache sharing*

Not true in modern systems

- Intervening in another cache more expensive than getting from memory

Cache-to-cache sharing also adds complexity

- How does memory know it should supply data (must wait for caches)
- Selection algorithm if multiple caches have valid data

But valuable for cache-coherent machines with distributed memory

- May be cheaper to obtain from nearby cache than distant memory
- Especially when constructed out of SMP nodes (Stanford DASH)

Dragon Write-back Update Protocol

4 states

- Exclusive-clean or exclusive (E): I and memory have it
- Shared clean (Sc): I, others, and maybe memory, but I'm not owner
- Shared modified (Sm): I and others but not memory, and I'm the owner
  - Sm and Sc can coexist in different caches, with only one Sm
- Modified or dirty (D): I and, noone else

No invalid state

- If in cache, cannot be invalid
- If not present in cache, can view as being in not-present or invalid state

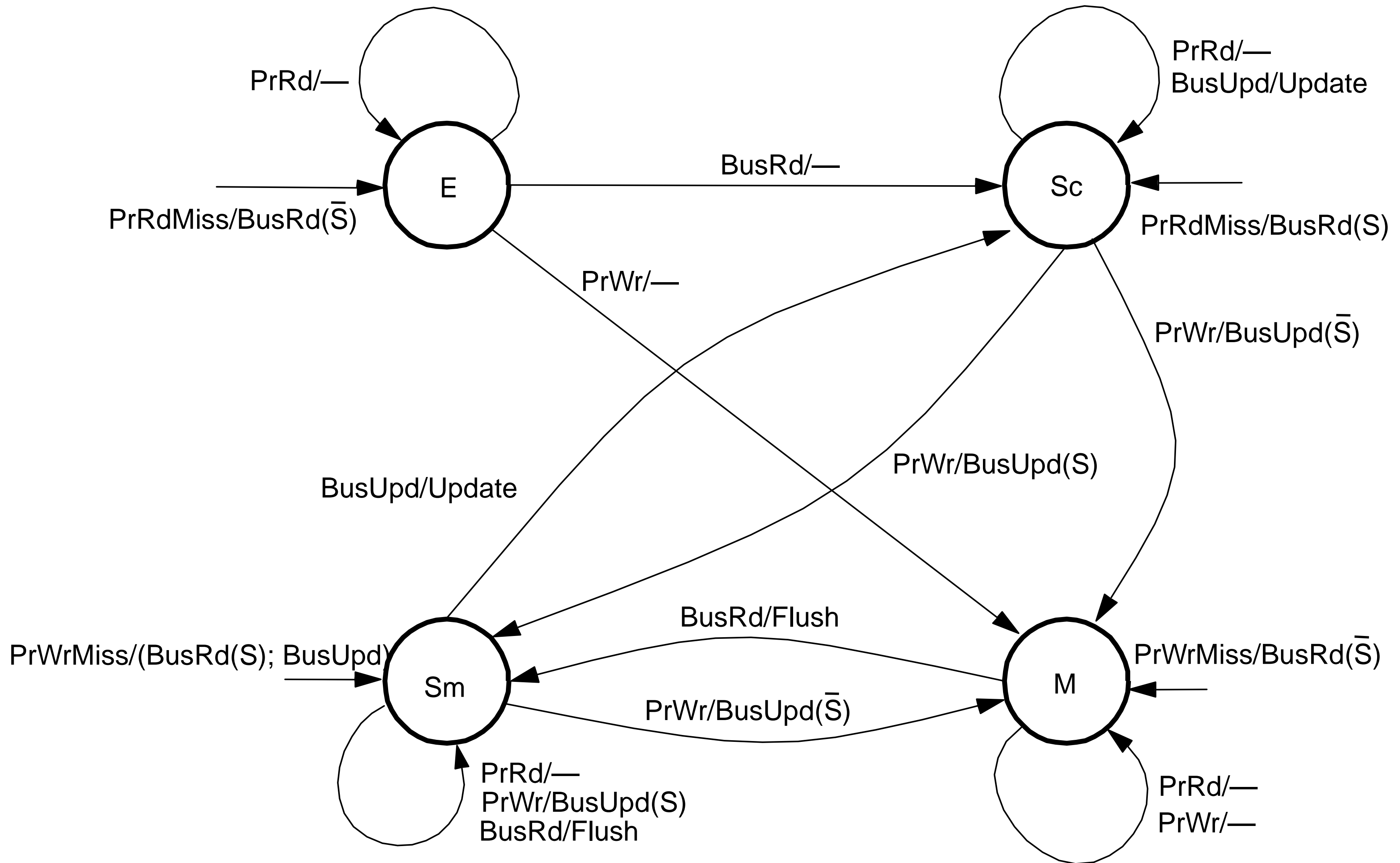
New processor events: PrRdMiss, PrWrMiss

- Introduced to specify actions when block not present in cache

New bus transaction: BusUpd

- Broadcasts single word written on bus; updates other relevant caches

Dragon State Transition Diagram



Lower-level Protocol Choices

Can shared-modified state be eliminated?

- If update memory as well on BusUpd transactions (DEC Firefly)
- Dragon protocol doesn't (assumes DRAM memory slow to update)

Should replacement of an Sc block be broadcast?

- Would allow last copy to go to E state and not generate updates
- Replacement bus transaction is not in critical path, later update may be

Shouldn't update local copy on write hit before controller gets bus

- Can mess up serialization

Coherence, consistency considerations much like write-through case

In general, many subtle race conditions in protocols

But first, let's illustrate quantitative assessment at logical level

Assessing Protocol Tradeoffs

Tradeoffs affected by performance and organization characteristics

Decisions affect pressure placed on these

Part art and part science

- Art: experience, intuition and aesthetics of designers
- Science: Workload-driven evaluation for cost-performance
  - want a balanced system: no expensive resource heavily underutilized

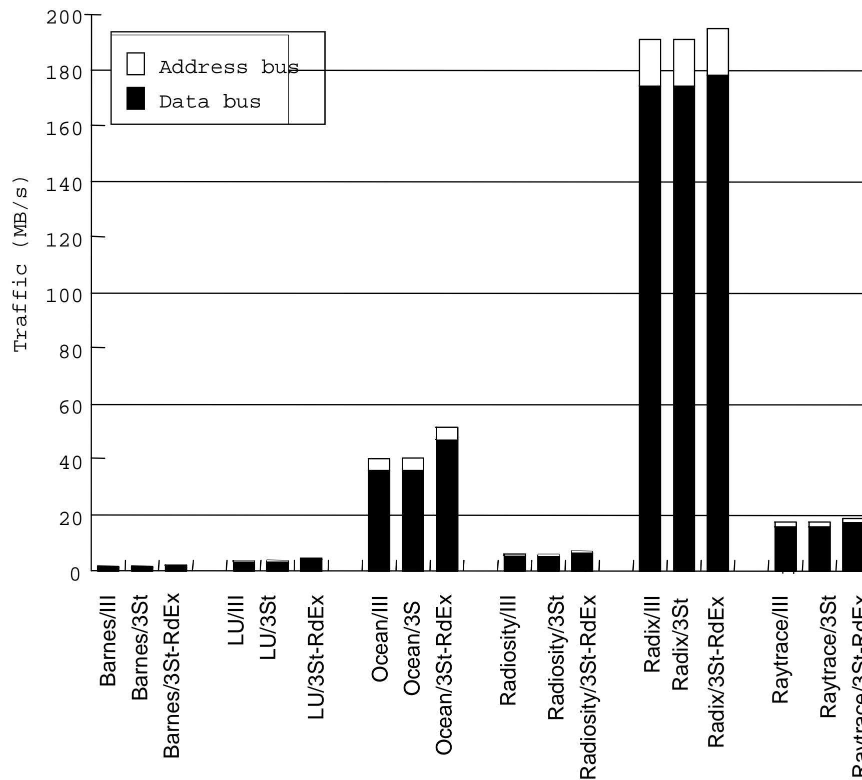
Methodology:

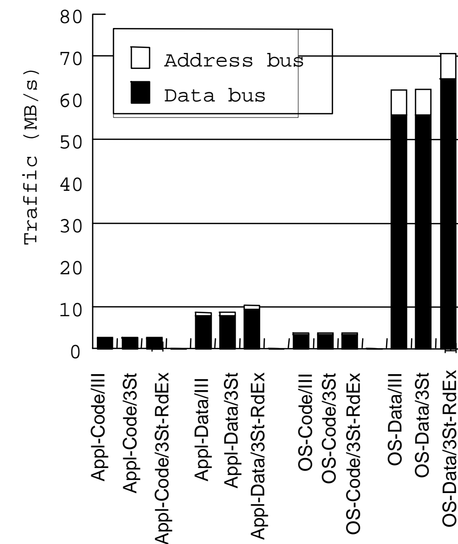
- Use simulator; choose parameters per earlier methodology (default 1MB, 4-way cache, 64-byte block, 16 processors; 64K cache for some)
- Focus on frequencies, not end performance for now
  - transcends architectural details, but not what we're really after
- Use idealized memory performance model to avoid changes of reference interleaving across processors with machine parameters
  - Cheap simulation: no need to model contention

Impact of Protocol Optimizations

(Computing traffic from state transitions discussed in book)

Effect of E state, and of BusUpgr instead of BusRdX





- MSI versus MESI doesn't seem to matter for bw for these workloads
- Upgrades instead of read-exclusive helps
- Same story when working sets don't fit for Ocean, Radix, Raytrace

Impact of Cache Block Size

Multiprocessors add new kind of miss to cold, capacity, conflict

- Coherence misses: true sharing and false sharing
  - latter due to granularity of coherence being larger than a word
- Both miss rate and traffic matter

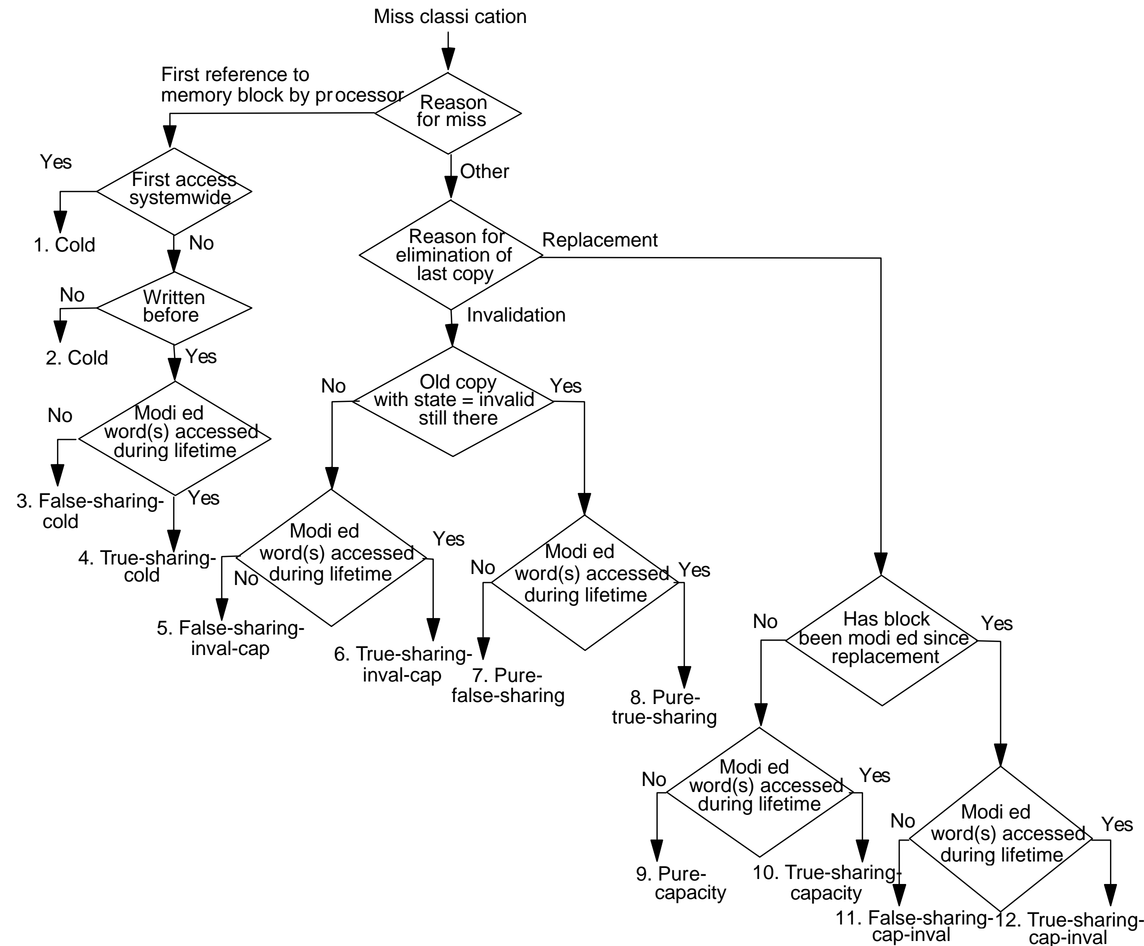
Reducing misses architecturally in invalidation protocol

- Capacity: enlarge cache; increase block size (if spatial locality)
- Conflict: increase associativity
- Cold and Coherence: only block size

Increasing block size has advantages and disadvantages

- Can reduce misses if spatial locality is good
- Can hurt too
  - increase misses due to false sharing if spatial locality not good
  - increase misses due to conflicts in fixed-size cache
  - increase traffic due to fetching unnecessary data and due to false sharing
  - can increase miss penalty and perhaps hit cost

A Classification of Cache Misses

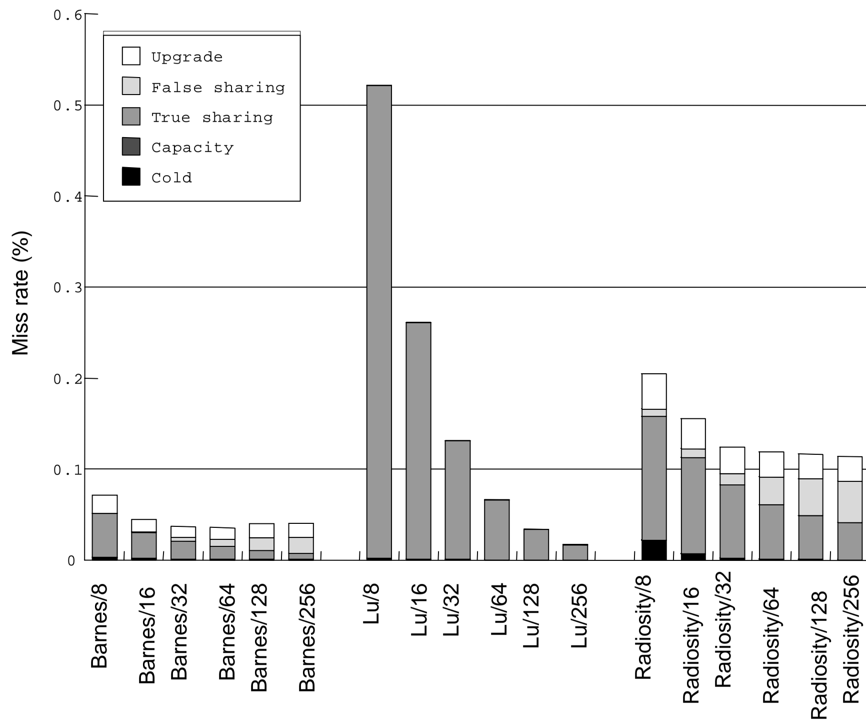


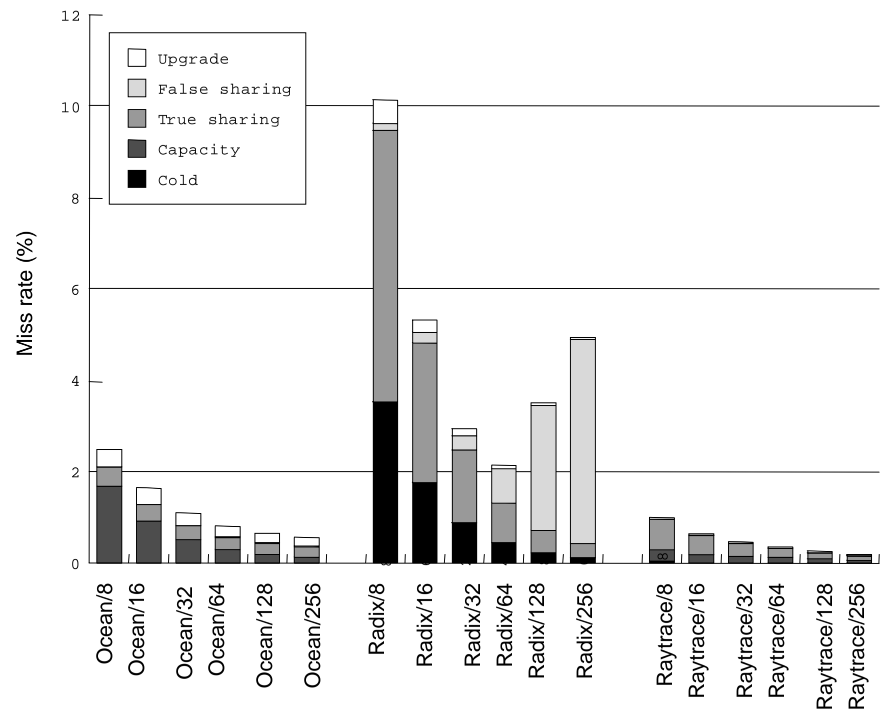
- Many mixed categories because a miss may have multiple causes

Impact of Block Size on Miss Rate

Results shown only for default problem size: varied behavior

- Need to examine impact of problem size and *p* as well (see text)

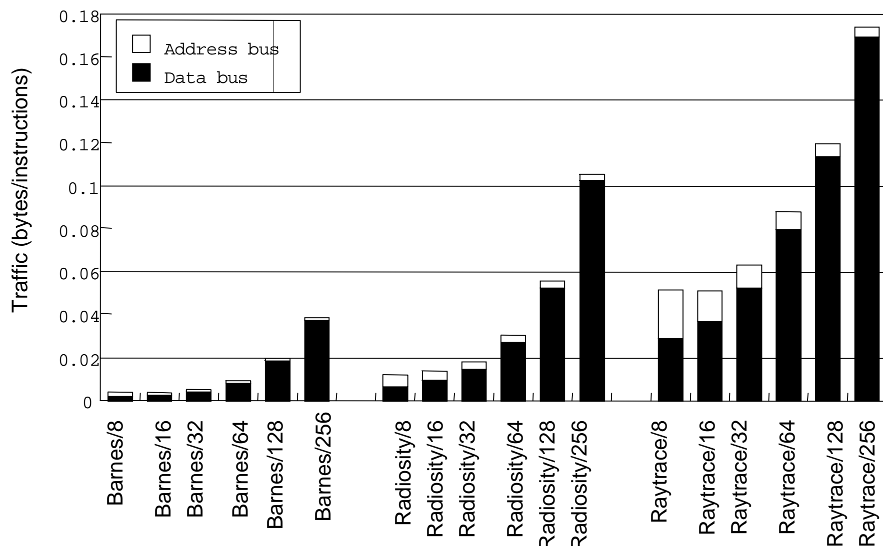


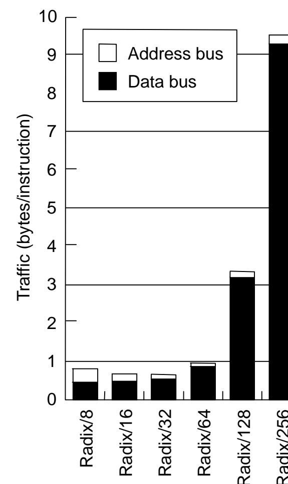


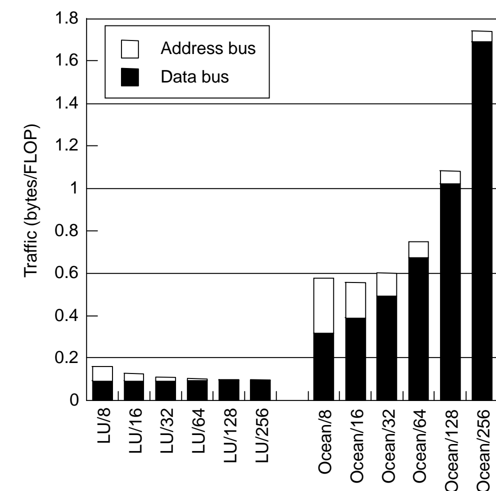
- Working set doesn't fit: impact on capacity misses much more critical

Impact of Block Size on Traffic

Traffic affects performance indirectly through contention







- Results different than for miss rate: traffic almost always increases
- When working sets fits, overall traffic still small, except for Radix
- Fixed overhead is significant component
  - So total traffic often minimized at 16-32 byte block, not smaller
- Working set doesn’t fit: even 128-byte good for Ocean due to capacity

Making Large Blocks More Effective

Software

- Improve spatial locality by better data structuring (more later)
- Compiler techniques

Hardware

- Retain granularity of transfer but reduce granularity of coherence
  - use subblocks: same tag but different state bits
  - one subblock may be valid but another invalid or dirty
- Reduce both granularities, but prefetch more blocks on a miss
- Proposals for adjustable cache size
- More subtle: delay propagation of invalidations and perform all at once
  - But can change consistency model: discuss later in course
- Use update instead of invalidate protocols to reduce false sharing effect

Update versus Invalidate

Much debate over the years: tradeoff depends on sharing patterns

Intuition:

- If those that used continue to use, and writes between use are few, update should do better
  - e.g. producer-consumer pattern
- If those that use unlikely to use again, or many writes between reads, updates not good
  - "pack rat" phenomenon particularly bad under process migration
  - useless updates where only last one will be used

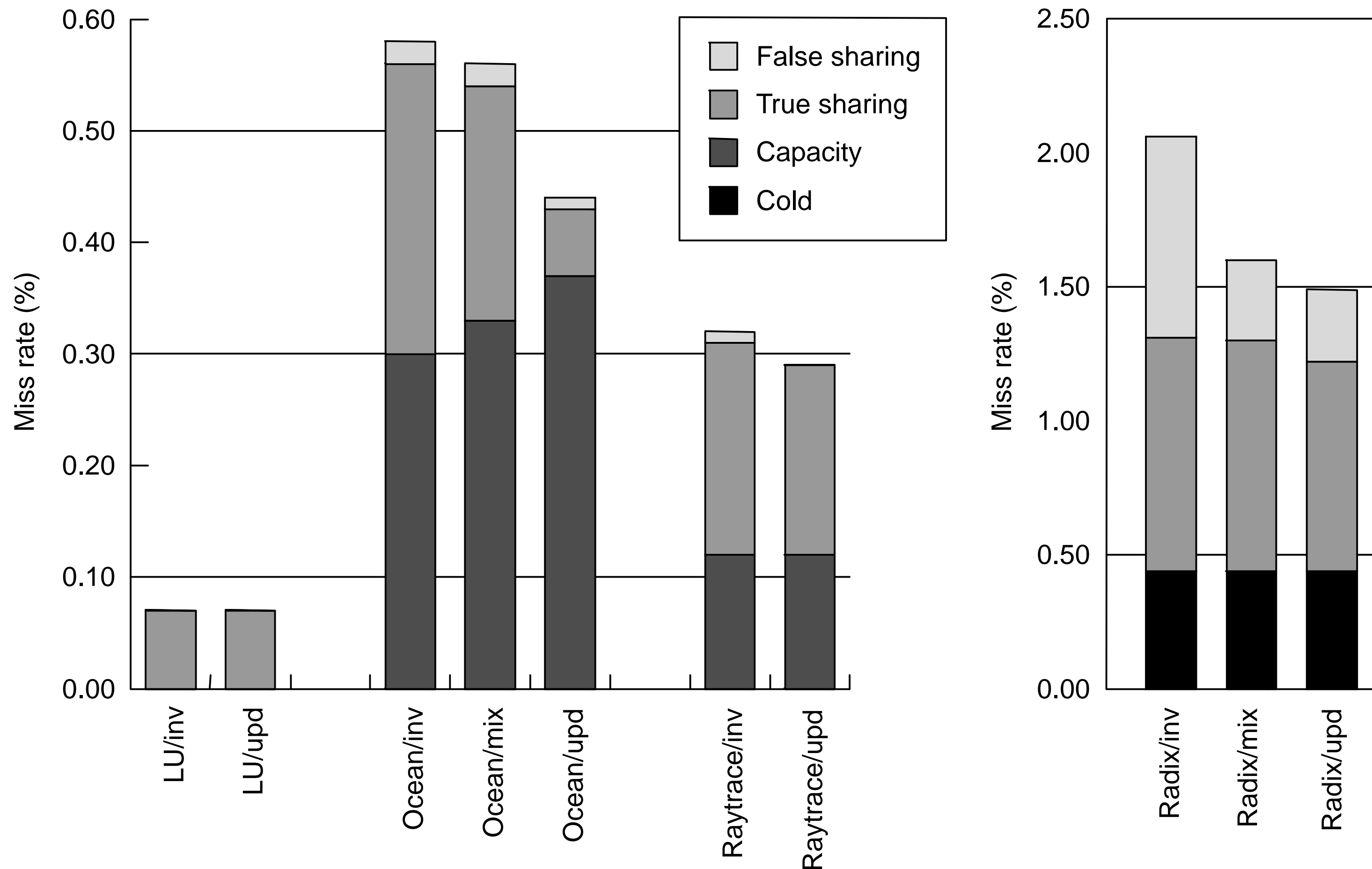
Can construct scenarios where one or other is much better

Can combine them in hybrid schemes (see text)

- E.g. competitive: observe patterns at runtime and change protocol

Let's look at real workloads

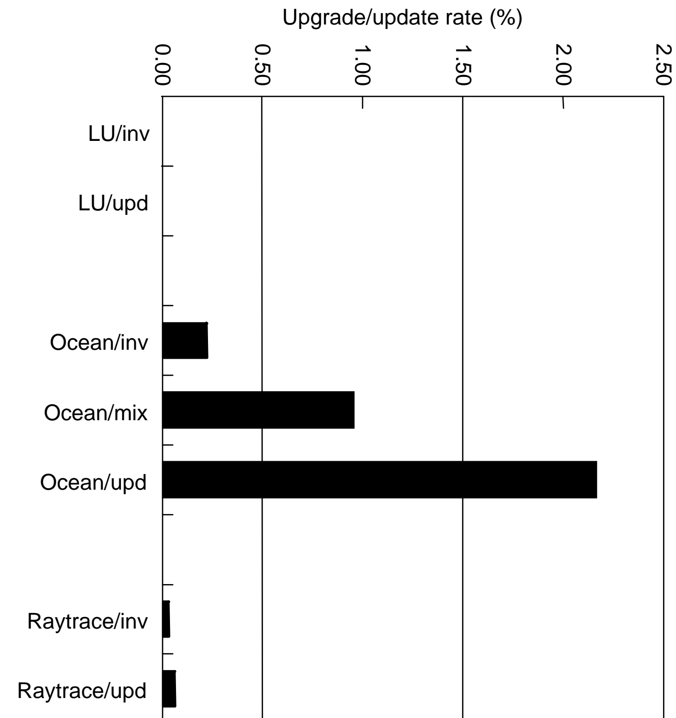
Update vs Invalidate: Miss Rates

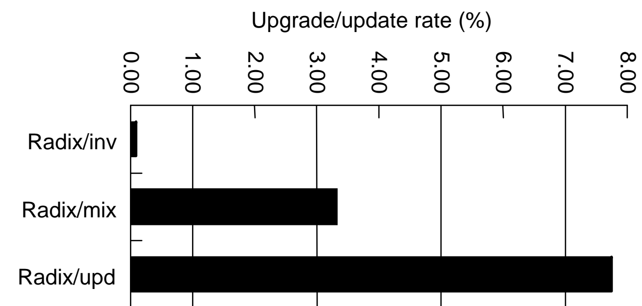


- Lots of coherence misses: updates help
- Lots of capacity misses: updates hurt (keep data in cache uselessly)
- Updates seem to help, but this ignores upgrade and update traffic

Upgrade and Update Rates (Traffic)

- Update traffic is substantial
- Main cause is multiple writes by a processor before a read by other
  - many bus transactions versus one in invalidation case
  - could delay updates or use merging
- Overall, trend is away from update based protocols as default
  - bandwidth, complexity, large blocks trend, pack rat for process migration
- Will see later that updates have greater problems for scalable systems





Synchronization

"A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."

Types of Synchronization

- *Mutual Exclusion*
- Event synchronization
  - *point-to-point*
  - group
  - *global (barriers)*

History and Perspectives

Much debate over hardware primitives over the years

Conclusions depend on technology and machine style

- speed vs flexibility

Most modern methods use a form of atomic read-modify-write

- IBM 370: included atomic compare&swap for multiprogramming
- x86: any instruction can be prefixed with a lock modifier
- High-level language advocates want hardware locks/barriers
  - but it's goes against the "RISC" flow,and has other problems
- SPARC: atomic register-memory ops (swap, compare&swap)
- MIPS, IBM Power: no atomic operations but pair of instructions
  - load-locked, store-conditional
  - later used by PowerPC and DEC Alpha too

Rich set of tradeoffs

Components of a Synchronization Event

Acquire method

- Acquire right to the synch (enter critical section, go past event

Waiting algorithm

- Wait for synch to become available when it isn't

Release method

- Enable other processors to acquire right to the synch

Waiting algorithm is independent of type of synchronization

Waiting Algorithms

Blocking

- Waiting processes are descheduled
- High overhead
- Allows processor to do other things

Busy-waiting

- Waiting processes repeatedly test a location until it changes value
- Releasing process sets the location
- Lower overhead, but consumes processor resources
- Can cause network traffic

Busy-waiting better when

- Scheduling overhead is larger than expected wait time
- Processor resources are not needed for other tasks
- Scheduler-based blocking is inappropriate (e.g. in OS kernel)

Hybrid methods: busy-wait a while, then block

Role of System and User

User wants to use high-level synchronization operations

- Locks, barriers...
- Doesn't care about implementation

System designer: how much hardware support in implementation?

- Speed versus cost and flexibility
- Waiting algorithm difficult in hardware, so provide support for others

Popular trend:

- System provides simple hardware primitives (atomic operations)
- Software libraries implement lock, barrier algorithms using these
- But some propose and implement full-hardware synchronization

Challenges

Same synchronization may have different needs at different times

- Lock accessed with low or high contention
- Different performance requirements: low latency or high throughput
- Different algorithms best for each case, and need different primitives

Multiprogramming can change synchronization behavior and needs

- Process scheduling and other resource interactions
- May need more sophisticated algorithms, not so good in dedicated case

Rich area of software-hardware interactions

- Which primitives available affects what algorithms can be used
- Which algorithms are effective affects what primitives to provide

Need to evaluate using workloads

Mutual Exclusion: Hardware Locks

Separate lock lines on the bus: holder of a lock asserts the line

- Priority mechanism for multiple requestors

Lock registers (Cray XMP)

- Set of registers shared among processors

Inflexible, so not popular for general purpose use

  - few locks can be in use at a time (one per lock line)
  - hardwired waiting algorithm

Primarily used to provide atomicity for higher-level software locks

First Attempt at Simple Software Lock

```
lock:       ld    register, location    /* copy location to register */
            cmp   location, #0          /* compare with 0 */
            bnz   lock                  /* if not 0, try again */
            st    location, #1 /        * store 1 to mark it locked */
            ret                         /* return control to caller */
```

and

```
unlock:     st    location, #0          /* write 0 to location */
            ret                         /* return control to caller */
```

Problem: lock needs atomicity in its own implementation

- Read (test) and write (set) of lock variable by a process not atomic

Solution: *atomic read-modify-write* or *exchange* instructions

- atomically test value of location and set it to another value, return success or failure somehow

Atomic Exchange Instruction

Specifies a location and register. In atomic operation:

- Value in location read into a register
- Another value (function of value read or not) stored into location

Many variants

- Varying degrees of flexibility in second part

Simple example: test&set

- Value in location read into a specified register
- Constant 1 stored into location
- Successful if value loaded into register is 0
- Other constants could be used instead of 1 and 0

Can be used to build locks

Simple Test&Set Lock

```
lock:      t&s   register, location
           bnz   lock                 /* if not 0, try again */
           ret                        /* return control to caller */
unlock:    st    location, #0         /* write 0 to location */
           ret                        /* return control to caller */
```

Other read-modify-write primitives can be used too

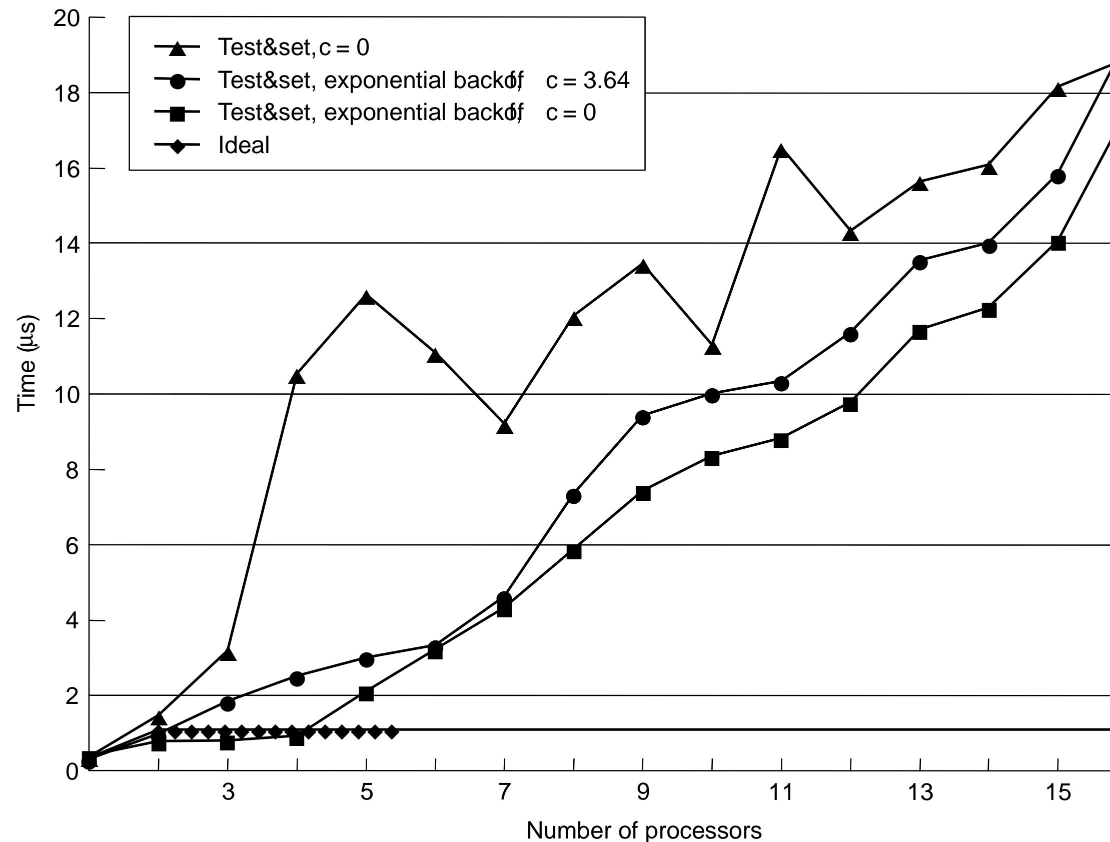
- Swap
- Fetch&op
- Compare&swap
  - Three operands: location, register to compare with, register to swap with
  - Not commonly supported by RISC instruction sets

Can be cacheable or uncacheable (we assume cacheable)

T&S Lock Microbenchmark Performance

On SGI Challenge. Code: `lock; delay(c); unlock;`

Same total no. of lock calls as p increases; measure time per transfer



- Performance degrades because unsuccessful test&sets generate traffic

Enhancements to Simple Lock Algorithm

Reduce frequency of issuing test&sets while waiting

- *Test&set lock with backoff*
- Don't back off too much or will be backed off when lock becomes free
- Exponential backoff works quite well empirically: i^{th} time = $k*c^i$

Busy-wait with read operations rather than test&set

- *Test-and-test&set lock*
- Keep testing with ordinary load
  - cached lock variable will be invalidated when release occurs
- When value changes (to 0), try to obtain lock with test&set
  - only one attemptor will succeed; others will fail and start testing again

Performance Criteria (T&S Lock)

Uncontended Latency

- Very low if repeatedly accessed by same processor; indept. of *p*

Traffic

- Lots if many processors compete; poor scaling with *p*
- Each t&s generates invalidations, and all rush out again to t&s

Storage

- Very small (single variable); independent of *p*

Fairness

- Poor, can cause starvation

Test&set with backoff similar, but less traffic

Test-and-test&set: slightly higher latency, much less traffic

But still all rush out to read miss and test&set on release

- Traffic for *p* processors to access once each: $O(p^2)$

Luckily, better hardware primitives as well as algorithms exist

Improved Hardware Primitives: LL-SC

Goals:

- Test with reads
- Failed read-modify-write attempts don't generate invalidations
- Nice if single primitive can implement range of r-m-w operations

Load-Locked (or -linked), *Store-Conditional*

LL reads variable into register

Follow with arbitrary instructions to manipulate its value

SC tries to store back to location if and only if no one else has written to the variable since this processor's LL

- If SC succeeds, means all three steps happened atomically
- If fails, doesn't write or generate invalidations (need to retry LL)
- Success indicated by condition codes; implementation later

Simple Lock with LL-SC

```
lock:     ll    reg1, location    /* LL location to reg1 */
          sc    location, reg2    /* SC reg2 into location*/
          beqz  reg2, lock        /* if failed, start again */
          ret
unlock:   st    location, #0      /* write 0 to location */
          ret
```

Can do more fancy atomic ops by changing what's between LL & SC

- But keep it small so SC likely to succeed
- Don't include instructions that would need to be undone (e.g. stores)

SC can fail (without putting transaction on bus) if:

- Detects intervening write even before trying to get bus
- Tries to get bus but another processor's SC gets bus first

LL, SC are not lock, unlock respectively

- Only guarantee no conflicting write to lock variable between them
- But can use directly to implement simple operations on shared variables

More Efficient SW Locking Algorithms

Problem with Simple LL-SC lock

- No invals on failure, but read misses by all waiters after both release and successful SC by winner
- No test-and-test&set analog, but can use backoff to reduce burstiness
- Doesn't reduce traffic to minimum, and not a fair lock

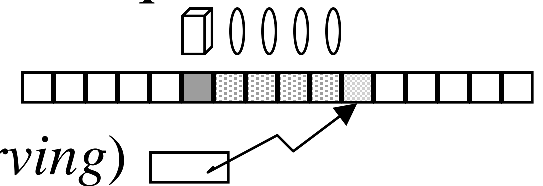
Better SW algorithms for bus (for r-m-w instructions or LL-SC)

- Only one process to try to get lock upon release
  - valuable when using test&set instructions; LL-SC does it already
- Only one process to have read miss upon release
  - valuable with LL-SC too
- *Ticket lock* achieves first
- *Array-based queueing lock* achieves both
- Both are fair (FIFO) locks as well

Ticket Lock

Only one r-m-w (from only one processor) per acquire

Works like waiting line at deli or bank



- Two counters per lock (*next_ticket*, *now_serving*)
- Acquire: fetch&inc *next_ticket;* wait for *now_serving* to equal it
  – atomic op when arrive at lock, not when it's free (so less contention)
- Release: increment *now-serving*
- FIFO order, low latency for low-contention if fetch&inc cacheable
- Still *O(p)* read misses at release, since all spin on same variable
  – like simple LL-SC lock, but no inval when SC succeeds, and fair
- Can be difficult to find a good amount to delay on backoff
  – exponential backoff not a good idea due to FIFO order
  – backoff proportional to *now-serving - next-ticket* may work well

Wouldn't it be nice to poll different locations ...

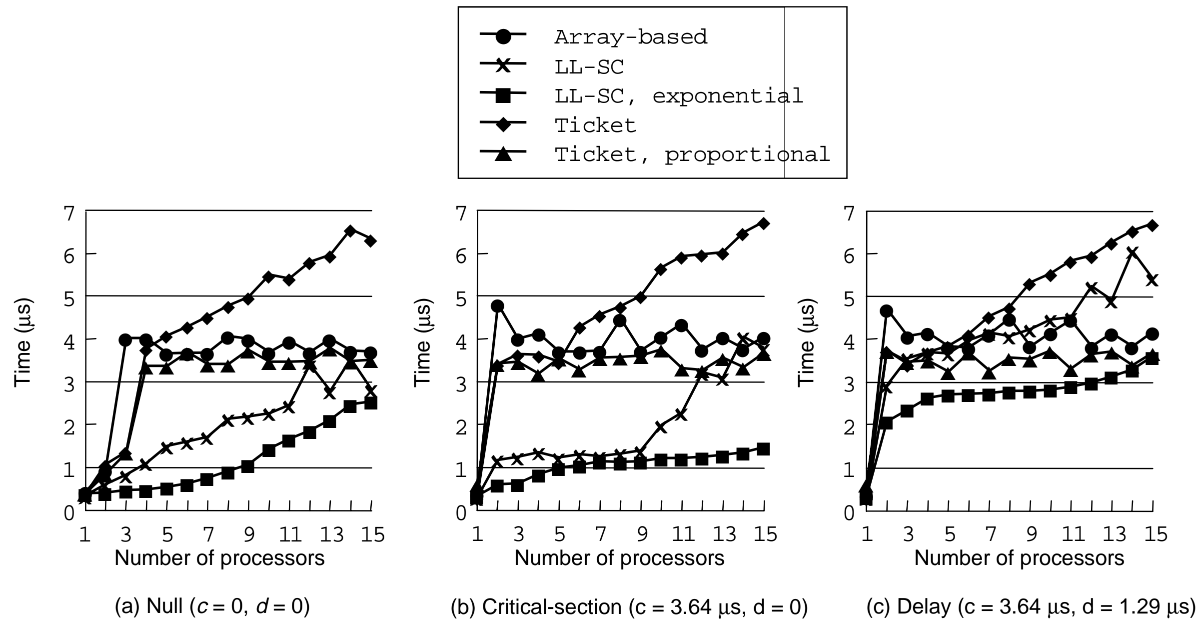
Array-based Queuing Locks

Waiting processes poll on different locations in an array of size p

- Acquire
  - fetch&inc to obtain address on which to spin (next array element)
  - ensure that these addresses are in different cache lines or memories
- Release
  - set next location in array, thus waking up process spinning on it
- *O(1)* traffic per acquire with coherent caches
- FIFO ordering, as in ticket lock
- But, *O(p)* space per lock
- Good performance for bus-based machines
- Not so great for non-cache-coherent machines with distributed memory
  - array location I spin on not necessarily in my local memory (solution later)

Lock Performance on SGI Challenge

Loop: `lock; delay(c); unlock; delay(d);`



- Simple LL-SC lock does best at small p due to unfairness
  - Not so with delay between unlock and next lock
  - Need to be careful with backoff
- Ticket lock with proportional backoff scales well, as does array lock
- Methodologically challenging, and need to look at real workloads

Point to Point Event Synchronization

Software methods:

- Interrupts
- Busy-waiting: use ordinary variables as flags
- Blocking: use semaphores

Full hardware support: *full-empty bit* with each word in memory

- Set when word is "full" with newly produced data (i.e. when written)
- Unset when word is "empty" due to being consumed (i.e. when read)
- Natural for word-level producer-consumer synchronization
  - producer: write if empty, set to full; consumer: read if full; set to empty
- Hardware preserves atomicity of bit manipulation with read or write
- Problem: flexiblity
  - multiple consumers, or multiple writes before consumer reads?
  - needs language support to specify when to use
  - composite data structures?

Barriers

Software algorithms implemented using locks, flags, counters

Hardware barriers

- Wired-AND line separate from address/data bus
- Set input high when arrive, wait for output to be high to leave
- In practice, multiple wires to allow reuse
- Useful when barriers are global and very frequent
- Difficult to support arbitrary subset of processors
  - even harder with multiple processes per processor
- Difficult to dynamically change number and identity of participants
  - e.g. latter due to process migration
- Not common today on bus-based machines

Let’s look at software algorithms with simple hardware primitives

A Simple Centralized Barrier

Shared counter maintains number of processes that have arrived

- increment when arrive (lock), check until reaches numprocs

```
struct bar_type {int counter; struct lock_type lock; int
 flag = 0;} bar_name;
BARRIER (bar_name, p) {
 LOCK(bar_name.lock);
 if (bar_name.counter == 0)
      bar_name.flag = 0;              /* reset flag if first to reach*/
 mycount = bar_name.counter++;        /* mycount is private */
 UNLOCK(bar_name.lock);
 if (mycount == p) {                  /* last to arrive */
      bar_name.counter = 0;           /* reset for next barrier */
      bar_name.flag = 1;              /* release waiters */
 }
 else while (bar_name.flag == 0) {};  /* busy wait for release */
}
```

- Problem?

A Working Centralized Barrier

Consecutively entering the same barrier doesn't work

- Must prevent process from entering until all have left previous instance
- Could use another counter, but increases latency and contention

Sense reversal: wait for flag to take different value consecutive times

- Toggle this value only when all processes reach

```
BARRIER (bar_name, p) {
  local_sense = !(local_sense); /* toggle private sense variable */
  LOCK(bar_name.lock);
  mycount = bar_name.counter++;       /* mycount is private */
  if (bar_name.counter == p)
        UNLOCK(bar_name.lock);
        bar_name.flag = local_sense;   /* release waiters*/
  else
      { UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {}; }
}
```

Centralized Barrier Performance

Latency

- Want short critical path in barrier
- Centralized has critical path length at least proportional to p

Traffic

- Barriers likely to be highly contended, so want traffic to scale well
- About $3p$ bus transactions in centralized

Storage Cost

- Very low: centralized counter and flag

Fairness

- Same processor should not always be last to exit barrier
- No such bias in centralized

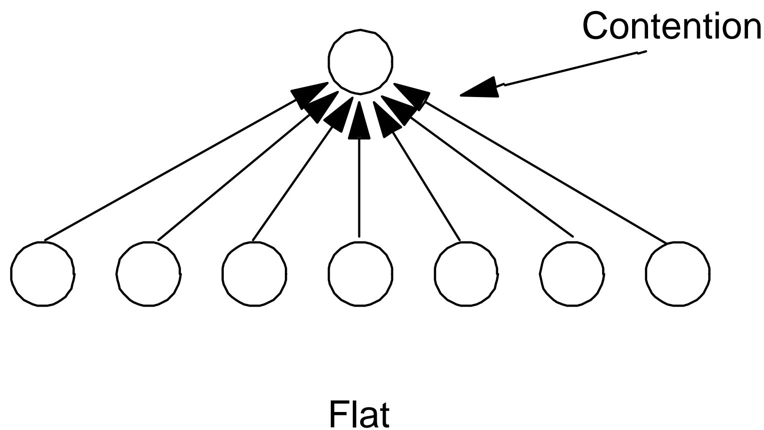
Key problems for centralized barrier are latency and traffic

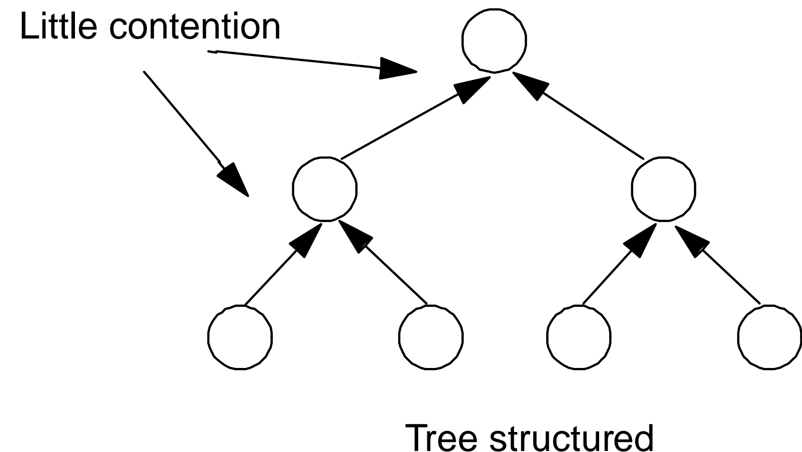
- Especially with distributed memory, traffic goes to same node

Improved Barrier Algorithms for a Bus

Software combining tree

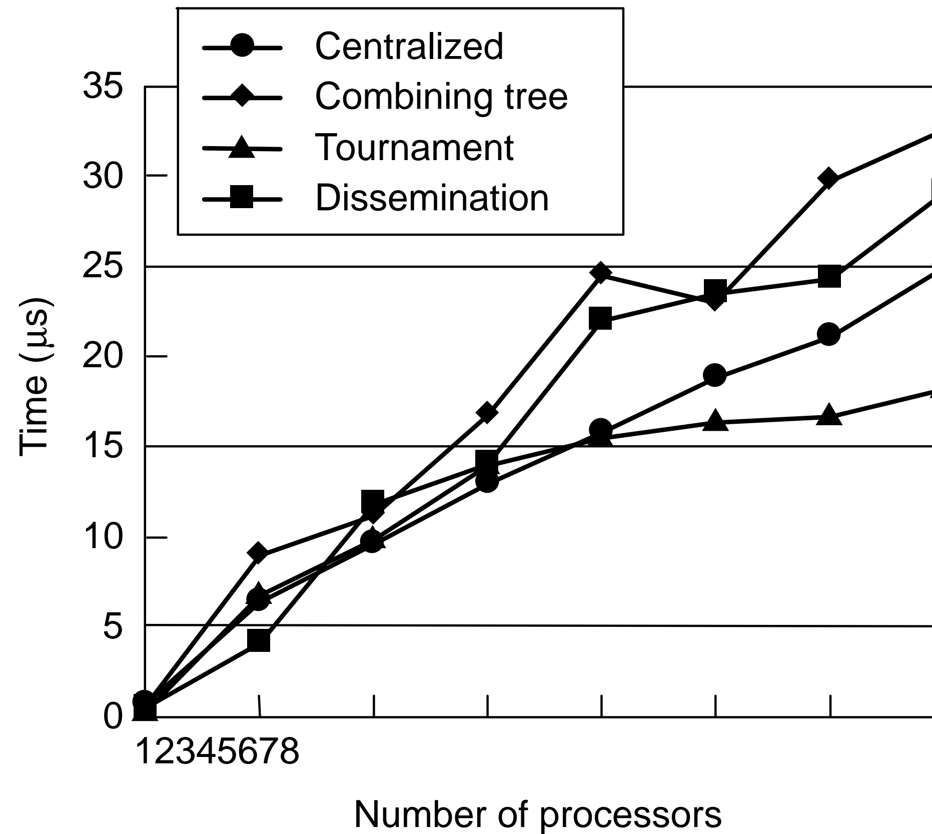
- Only k processors access the same location, where k is degree of tree





- Separate arrival and exit trees, and use sense reversal
- Valuable in distributed network: communicate along different paths
- On bus, all traffic goes on same bus, and no less total traffic
- Higher latency (*log p* steps of work, and *O(p)* serialized bus xactions)
- Advantage on bus is use of ordinary reads/writes instead of locks

Barrier Performance on SGI Challenge



- Centralized does quite well
  - Will discuss fancier barrier algorithms for distributed machines
- Helpful hardware support: piggybacking of reads misses on bus
  - Also for spinning on highly contended locks

Synchronization Summary

Rich interaction of hardware-software tradeoffs

Must evaluate hardware primitives and software algorithms together

- primitives determine which algorithms perform well

Evaluation methodology is challenging

- Use of delays, microbenchmarks
- Should use both microbenchmarks and real workloads

Simple software algorithms with common hardware primitives do well on bus

- Will see more sophisticated techniques for distributed machines
- Hardware support still subject of debate

Theoretical research argues for swap or compare&swap, not fetch&op

- Algorithms that ensure constant-time access, but complex

Implications for Parallel Software

Looked at how software affects architecture; now do reverse

Load balance, inherent comm. and extra work issues same as before

- Also, assign so that one processor writes a set of data, at least in a phase
- e.g. in graphics, usually partition image rather than scene

Structure of communication and mapping are not major issues

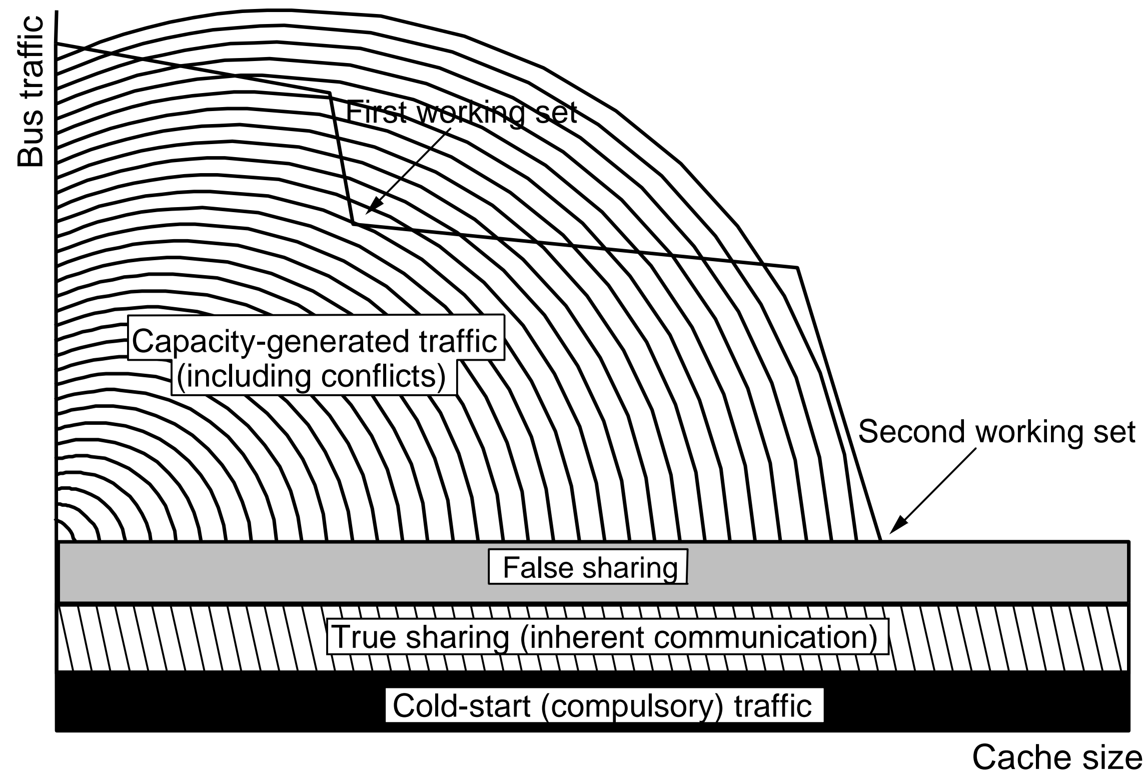
Key is temporal and spatial locality in orchestration step

- Reduce misses and hence both latency and traffic
- Temporal locality: keep working sets tight enough to fit in cache
- Spatial locality: reduce fragmentation and false sharing

Temporal Locality

Main memory centralized, so exploit in processor caches

Specialization of general working set curve for buses



- Techniques same as discussed earlier for general case

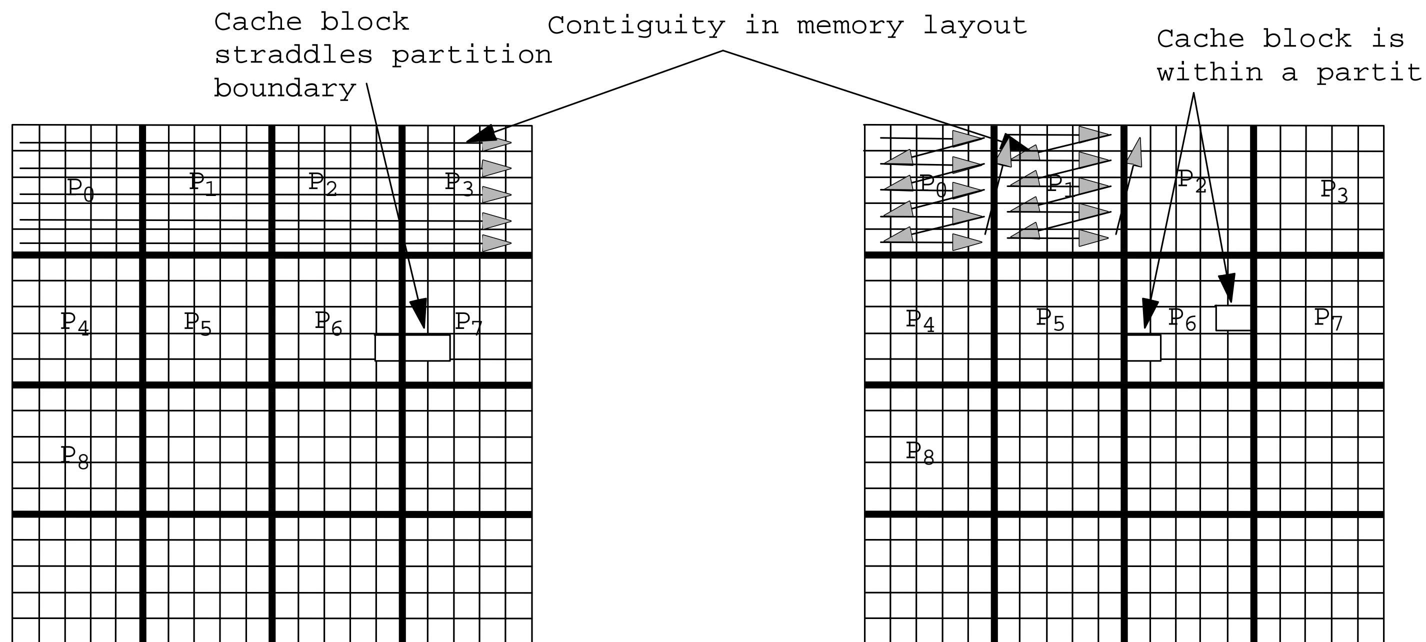
Bag of Tricks for Spatial Locality

Assign tasks to reduce spatial interleaving of accesses from procs

- Contiguous rather than interleaved assignment of array elements

Structure data to reduce spatial interleaving of accesses from procs

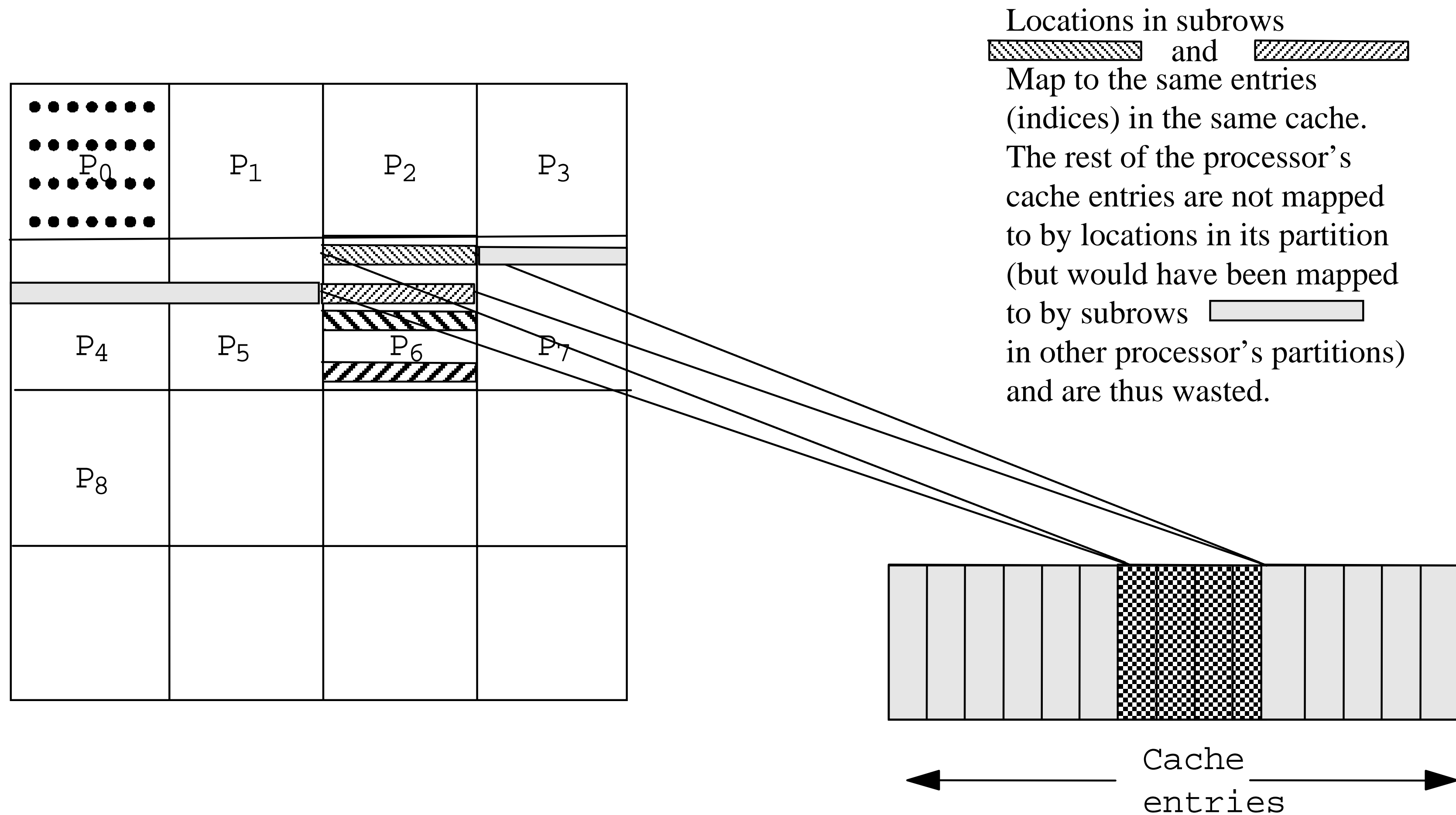
- Higher-dimensional arrays to keep partitions contiguous
- Reduce false sharing and fragmentation as well as conflict misses



(a) Two-dimensional array

(b) Four-dimensional array

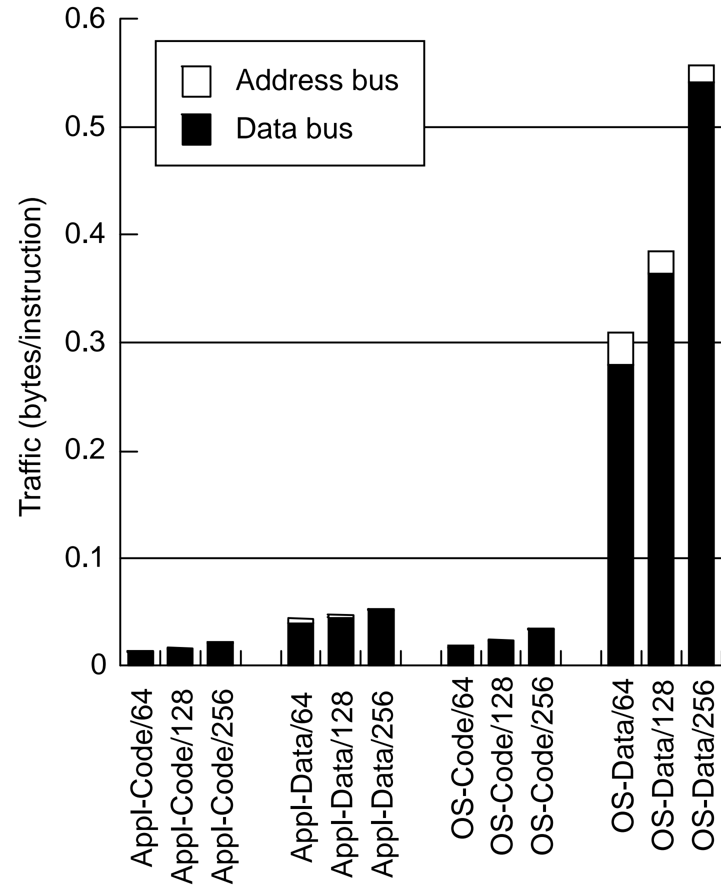
Conflict Misses in a 2-D Array Grid



- Consecutive subrows of partition are not contiguous
- Especially problematic when both array and cache size is power of 2

Performance Impact

Performance on 16-processor SGI Challenge



- Impact of false sharing and conflict misses with 2D arrays clear

Bag of Tricks (contd.)

Beware conflict misses more generally

- Allocate non-power-of-2 even if application needs power-of-2
- Conflict misses across data structures: ad-hoc padding/alignment
- Conflict misses on small, seemingly harmless data

Use per-processor heaps for dynamic memory allocation

Copy data to increase locality

- If noncontiguous data are to be reused a lot, e.g. blocks in 2D-array LU
- Must trade off against cost of copying

Pad and align arrays: can have false sharing v. fragmentation tradeoff

Organize arrays of records for spatial locality

- E.g. particles with fields: organize by particle or by field
- In vector programs by field for unit-stride, in parallel often by particle
- Phases of program may have different access patterns and needs

These issues can have greater impact than inherent communication

- Can cause us to revisit assignment decisions (e.g. strip v. block in grid)

Concluding Remarks

SMPs are natural extension of uniprocessors, increasingly popular

- Graceful path for parallelization
- Fine-grained sharing for multiprogramming and OS

Key technical challenge is design of extended memory hierarchy

- Many tradeoffs in bus and protocol design even at logical level

Should continue to be important

- Attractive cost-performance
- Microprocessors are multiprocessor-ready, so no time-lag
- Software technology maturing
- Attractive as nodes for larger parallel machine (cost amortization)
- Multiprocessor on a chip

Real action is at the next level of protocol and implementation

Shared Cache: Examples

Alliant FX-8

- Eight 68020s with crossbar to 512K interleaved cache
- Focus on bandwidth to shared cache and memory

Encore, Sequent

- Two processors (N32032) to a board with shared cache
- Cache-coherent bus across boards
- Amortize hardware overhead of coherence; slow processors

As transistors per chip increase, shared-cache on a chip?

Shared Cache Advantages

No need for coherence!

- Only one copy of any cached block

Fine-grained sharing

- Communication latency determined by where in hierarchy paths meet
- 2-10 cycles; as opposed to 20-150 cycles at shared memory

Processors prefetch data for one another

No false-sharing (ping-ponging)

Smaller total cache requirements

- Overlapping working sets

Shared Cache Disadvantages

Very high cache bandwidth requirements

Increased latency for all accesses (incl. hits!)

- Crossbar interconnect latency
- Large cache
- L1 cache hit time important determinant of processor cycle time!

Contention at cache

Negative interference (conflict or capacity)

Not currently supported by commodity microprocessors

List-based Queuing Locks

List-based locks

- build linked-lists per lock in SW
- acquire
  - allocate (local) list element and enqueue on list
  - spin on flag field of that list element
- release
  - set flag of next element on list
- use compare&swap to manage lists
  - swap is sufficient, but lose FIFO property
  - FIFO
  - spin locally (cache-coherent or not)
  - *O(1)* network transactions even without consistent caches
  - *O(1)* space per lock
  - but, compare&swap difficult to implement in hardware

Recent Areas of Investigation

Multi-protocol Synchronization Algorithms

- Reactive algorithms
- Adaptive waiting mechanisms
- Wait-free algorithms

Integration with OS scheduling

Multithreading

- what do you do while you wait?
  - could be much longer than a memory access

Implementing Atomic Ops with Caching

One possibility: Load Linked / Store Conditional (LL/SC)

- Load Linked loads the lock and sets a bit
- When "atomic" operation is done, Store Conditional succeeds only if bit was not reset in interim
- Doesn't need diff instructions with diff nos. of arguments
- Good for bus-based machine: SC result delivered by bus
- More complex for directory-based machine:
  - wait for SC to go to directory and get ownership (long latency)
  - have LL load in exclusive mode, so SC succeeds immediately if still in exclusive mode

Bottom Line for Locks

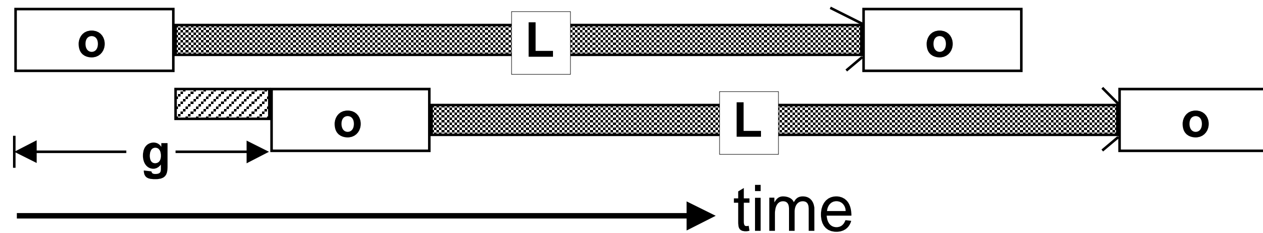
Lots of options

SW algorithms can do well given simple HW primitives (fetch&op)

- LL/SC works well if there is locality of synch access
- Otherwise, in-memory fetch&ops are good for high contention

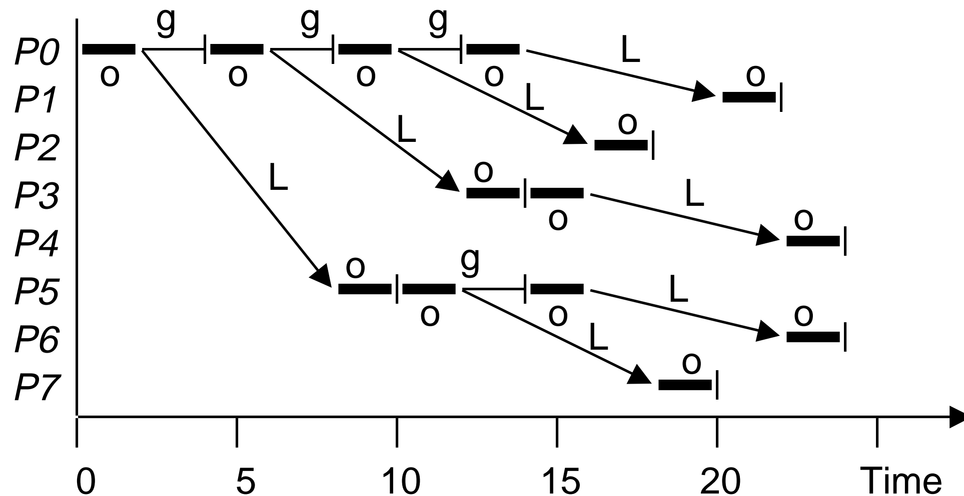
Optimal Broadcast

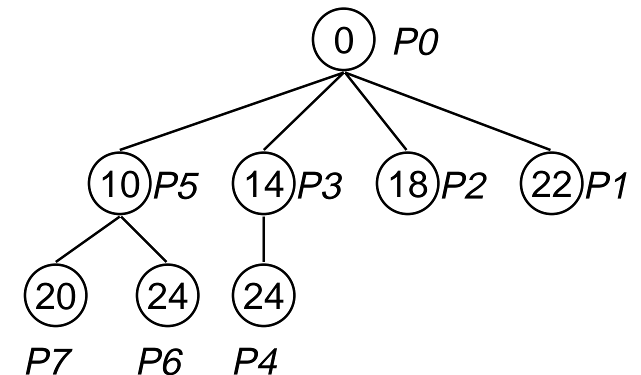
Model: Latency, Overhead, Gap



Optimal single item broadcast is an unbalanced tree

– shape determined by relative values of *L, o,* and *g.*





Dissemination Barrier

Goal is to allow statically allocated flags

- avoid remote spinning even without cache coherence

log p rounds of synchronization

In round k, proc i synchronizes with proc $(i+2^k) \bmod p$

- can statically allocate flags to avoid remote spinning

Like a butterfly network

Tournament Barrier

Like binary combining tree

But representative processor at a node chosen statically

- no fetch-and-op needed

In round k, proc i sets a flag for proc $j = i - 2^k \ (mod\ 2^{k+1})$

- i then drops out of tournament and j proceeds in next round
- i waits for global flag signalling completion of barrier to be set by root
  - could use combining wakeup tree

Without coherent caches and broadcast, suffers from either traffic due to single flag or same problem as combining trees (for wakeup)

MCS Barrier

Modifies tournament barrier to allow static allocation in wakeup tree, and to use sense reversal

Every processor is a node in two *p*-node trees

- has pointers to its parent, building a fanin-4 arrival tree
- has pointers to its children to build a fanout-2 wakeup tree

+ spins on local flag variables

+ requires *O(P)* space for *P* processors

+ theoretical minimum no. of network transactions *(2P -2)*

+ *O(log P)* network transactions on critical path

Recent Directions

Adaptive tree barriers

- late arrivals should be close to the root

Pipelined Scan Operations

Hardware Support ?

Space Requirements

Centralized:	constant
MCS, combining tree:	$O(p)$
Dissemination, Tournament:	$O(p \log p)$

Network Transactions

Centralized, combining tree:	$O(p)$ if broadcast and coherent caches; unbounded otherwise
Dissemination:	$O(p \log p)$
Tournament, MCS:	$O(p)$

Critical Path Length

If independent parallel network paths available:

- all are $O(\log P)$ except centralized, which is $O(P)$

If not (e.g. shared bus):

- linear terms dominate