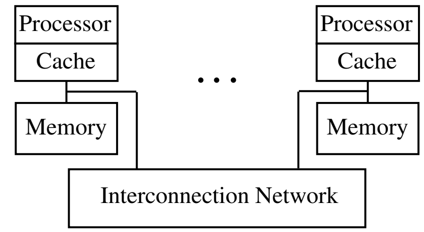
Memory Consistency Models for Shared-Memory Multiprocessors

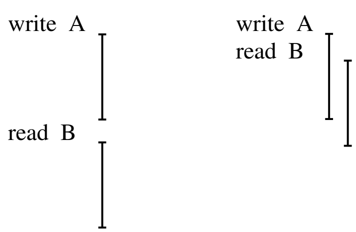
Motivation



- Build large-scale shared-memory multiprocessors
- High memory latency is a fundamental issue
  - Stanford DASH: 100-150 cycles
  - Kendall Square Research KSR1: 200-600 cycles
- Caches reduce latency, but inherent communication remains

Hiding Memory Latency

- Overlap memory accesses with other accesses and computation



- Simple in uniprocessors
- Can affect correctness in multiprocessors

Outline

- Memory Consistency Models
- Framework for Programming Simplicity
- Performance Evaluation
- Conclusions

Uniprocessor Memory Model

- Memory model specifies ordering constraints among accesses
- Uniprocessor model: memory accesses atomic and in program order

write A
write B
read A
read B

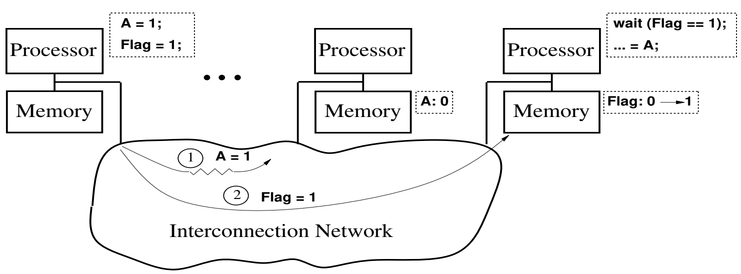
- Not necessary to maintain sequential order for correctness
  - hardware: buffering, pipelining
  - compiler: register allocation, code motion
- Simple for programmers
- Allows for high performance

Shared-Memory Multiprocessors

- Order between accesses to different locations becomes important

P1	P2
A = 1;	
Flag = 1;	wait (Flag == 1);
	... = A;

How Unsafe Reordering Can Happen

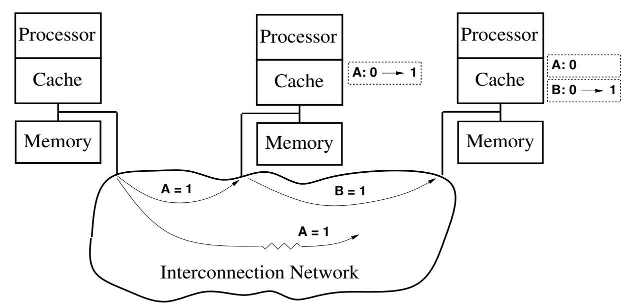


- Distribution of memory resources
  - accesses issued in order may be observed out of order

Caches Complicate Things More

- Multiple copies of the same location

P1	**P2**	**P3**
A = 1;	**wait (A==1);**	
	B = 1;	**wait (B==1);**
		... = A;



Need for a Multiprocessor Memory Model

• Provide reasonable ordering constraints on memory accesses

- affects programmers

- affects system designers

Memory Behavior

What should the semantics be for memory operations to the shared memory?

• ease-of-use: keep it similar to serial semantics for uniprocessor

• operating system community used concurrent programming:

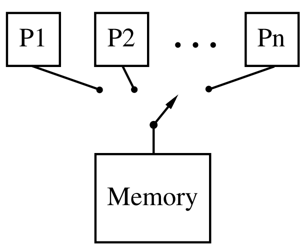
  • multiple processes interleaved on a single processor

• Lamport (1979) formalized **Sequential Consistency** (SC):

  • "... 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."

Sequential Consistency

• Formalized by Lamport
  - accesses of each processor in program order
  - all accesses appear in sequential order



• Any order implicitly assumed by programmer is maintained

Example with SC

Simple Synchronization:

P1	P2
A = 1 (a)	
Flag =1 (b)	x = Flag (c)
	y = A (d)

• all locations are initialized to 0

• possible outcomes for (x,y): (0,0), (0,1), (1,1)

• (x,y) = (1,0) is not a possible outcome:

  • we know a->b and c->d by program order
  • b->c implies that a->d
  • y==0 implies d->a which leads to a contradiction

Another Example with SC

From Dekker's Algorithm:

P1	P2
A = 1 (a)	B = 1 (c)
x = B (b)	y = A (d)

- possible outcomes for (x,y): (0,1), (1,0), (1,1)
- outcome (x,y) = (0,0) not possible:
  - a->b and c->d implied by program order
  - x = 0 implies b->c which implies a->d
  - a->d says y = 1 which leads to a contradiction
  - similarly, y = 0 implies x =1 which is also a contradiction

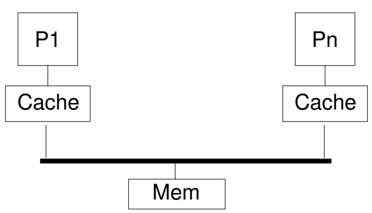
How to Guarantee SC

Sufficient Conditions for SC (Dubois et al., 1987):

- assumes general cache coherence (if we have caches):
  - writes to the same location are observed in same order by all P's
- for each processor, delay issue of access until previous completes
  - a read completes when the return value is back
  - a write completes when new value is visible to all processors
  - for simplicity, we assume writes are atomic

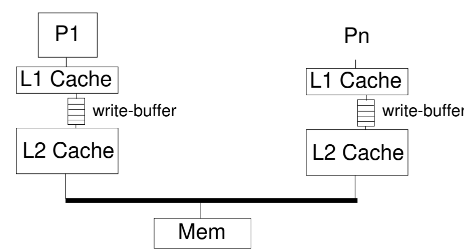
Important to note that these are not necessary conditions for maintaining SC

Simple Bus-Based Multiprocessor



- assume write-back caches
- general cache coherence maintained by serialization at bus
  - writes to same location serialized and observed in the same order by all
- writes are atomic because all processors observe the write at the same time
- accesses from a single process complete in program order:
  - cache is busy while servicing a miss, effectively delaying later access
- SC is guaranteed without any extra mechanism above coherence

Example of Complication in Bus-Based Machines



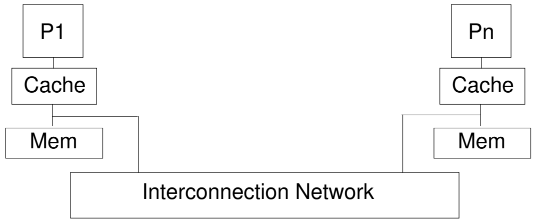
- 1st level cache write-thru, 2nd level write-back (e.g.,SGI cluster in DASH)
- write-buffer with no forwarding (reads to 2nd level delayed until buffer empty)

- never hit in the 1st level cache: SC is maintained (same as previous slide)
- read hits in the first level cache cause complication (e.g., Dekker's algorithm)

- to maintain SC, we need to delay access to 1st level until there are no writes pending in write buffer (full write latency observed by processor)

- multiprocessors may not maintain SC to achieve higher performance

Scalable Shared-Memory Multiprocessor



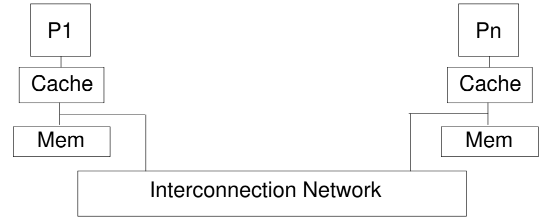
- no more bus to serialize accesses
- only order maintained by network is point-to-point

- general cache coherence:
  - serialize at memory location; point-to-point order required

- accesses issued in order do not necessarily complete in order:
  - due to distribution of memory and varied-length paths in network

- writes are inherently non-atomic:
  - new value is visible to some while others can still see old value
  - no one point in the system where a write is completed

Scalable Architectures (Cont'd)

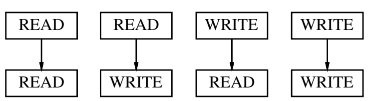


- need to know when a write completes:
  - for providing atomicity
  - for delaying an access until previous one completes
- requires acknowledgement messages:
  - write is complete when all invalidations are acknowledged
  - use a counter to count the number of acknowledgements
- ensuring atomicity for writes:
  - delay access to new value until all acknowledgments are back
  - can be done for invalidation-based schemes; unnatural for updates
- ensuring order of accesses from a processor:
  - delay each access until the previous one completes

- latencies are large (10's to 100's of cycles) and all latency seen by processor

Summary for Sequential Consistency

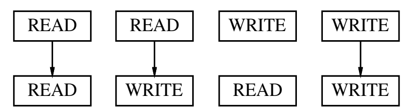
- Maintain order between shared accesses in each process



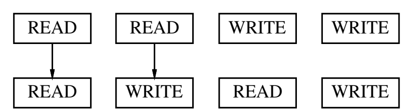
- Severely restricts common hardware and compiler optimizations

Alternatives to Sequential Consistency

- Relax constraints on memory order



Processor Consistency (PC)
Total Store Ordering (TSO)



Partial Store Ordering (PSO)

Relaxed Models

- Processor consistency (PC) - Goodman 89
- Total store ordering (TSO) - Sindhu 90
- Causal memory - Hutto 90
- PRAM - Lipton 90

- Partial store ordering (PSO) - Sindhu 90

- Weak ordering (WO) - Dubois 86

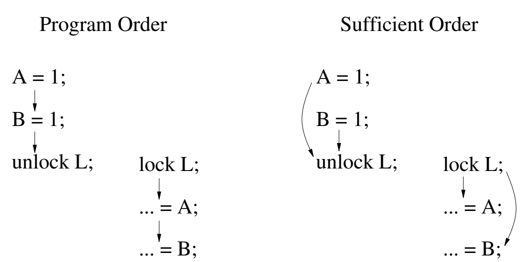
- Problems:
  - programming complexity
  - portability

Framework for Programming Simplicity

- Develop a unified framework that provides
  - programming simplicity
  - all previous performance optimizations, and more

Intuition

- “Correctness”: same results as sequential consistency
- Most programs don’t require strict ordering for “correctness”



- Difficult to automatically determine orders that are not necessary
- Specify methodology for writing “safe” programs

Overview of Framework

- Programmer
  - methodology for writing programs

- System designer
  - safe optimizations for such programs

Synchronized Programs

- Requirements:
  - all synchronizations are explicitly identified
  - all data accesses are ordered through synchronization

- How do synchronized programs get generated?

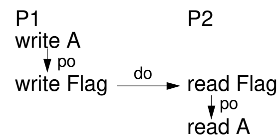
  1. Compiler generated parallel program
     - synchronization automatically identified

  2. Explicitly parallel program
     - easily identifiable synchronization constructs
     - programmer guarantees data access ordered

Identifying Data Races and Synchronization

Two accesses *conflict* if
- access same location
- at least one is a write

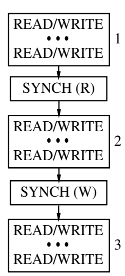
Order accesses by
- program order (po)
- dependence order (do): op1 --> op2 if op2 reads op1



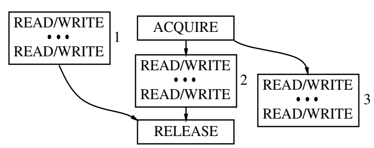
Data Race:
- two conflicting accesses on different processors
- not ordered by intervening accesses

Optimizations for Synchronized Programs

- Exploit information about synchronization



Weak Ordering (WO)



Release Consistency (RC)

- Proof: synchronized programs yield SC results on RC systems

Summary of Programmer Model

- Contract between programmer and system:
  - programmer provides synchronized programs
  - system provides sequential consistency at higher performance

- Allows portability over a wide range of implementations

- Research on similar frameworks:
  - Properly-labeled (PL) programs - Gharachorloo 90
  - Data-race-free (DRF) - Adve 90
  - Unifying framework (PLpc) - Gharachorloo,Adve 92

Outline

✓ Memory Consistency Models

✓ Framework for Programming Simplicity

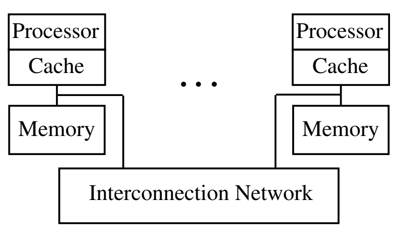
☞ Performance Evaluation

Performance Evaluation

• Goal: characterize gains from relaxed models

- relaxed models effective in hiding memory latency
- enhance gains from other latency hiding techniques

Architectural Assumptions



• Based on Stanford DASH multiprocessor

• Coherent caches, directory-based invalidation scheme

• Latency = 1:25:100 processor cycles

• Detailed simulation, contention modeled

• 16 processors

Benchmark Applications

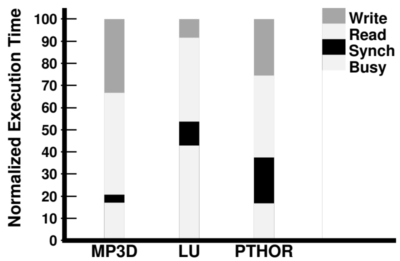
• MP3D: 3-dimensional particle simulator
  - 10,000 particles, 5 time steps

• LU: LU-decomposition of dense matrix
  - 200x200 matrix

• PTHOR: logic simulator
  - 11,000 two-input gates, 5 clock cycles

Performance of Sequential Consistency

• Processor issues accesses one-at-a-time and stalls for completion

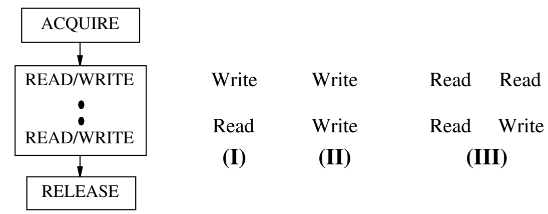


• Low processor utilization (17% - 42%) even with caching

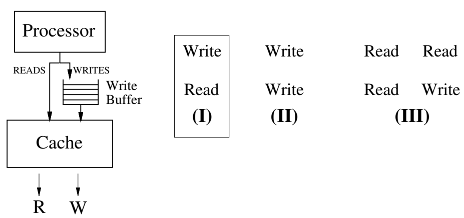
Relaxed Models

• Focus on release consistency

• Various degrees of aggressiveness:
  - implementation requirements
  - performance benefits



Requirement for W-R Overlap

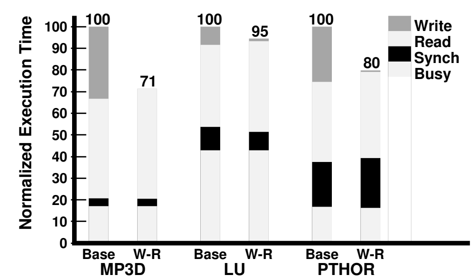


• Processor stalls on reads

• Reads bypass write buffer

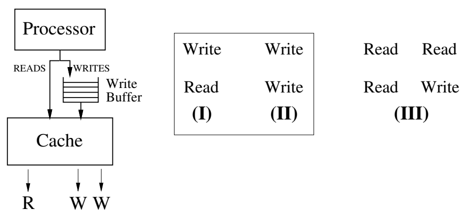
• Cache is lockup-free (Kroft 81)
  - allows more than one outstanding request

Allowing Reads to Overlap Previous Writes



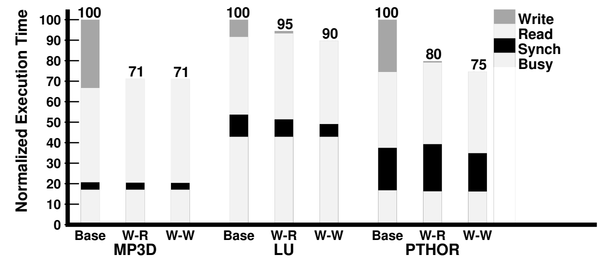
• Write latency is fully hidden

Requirement for W-W Overlap



• Cache allows multiple outstanding writes

Allowing Writes to Overlap



• Overlap of writes allows for faster synchronization on critical path

```
W A              LOCK L
W B              R A
UNLOCK L         R B
```

Requirement for R-R and R-W Overlap

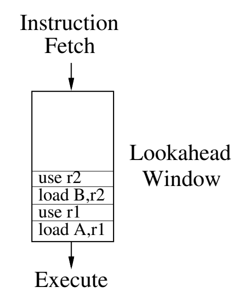
• Allow processor to continue past read misses

```
load  A, r1              load  A, r1
                         load B, r2

use r1                   use r1
load B, r2               use r2

use r2
```

• Lookahead ability provided in dynamically scheduled processor

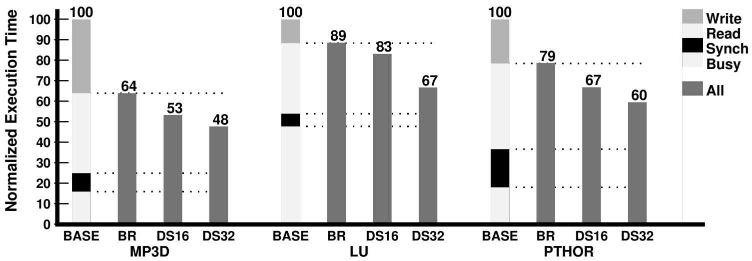
Dynamically Scheduled Processor

• Processor based on detailed design in Johnson's thesis



• Lookahead window size important

Results with Dynamically Scheduled Processor



BR: Blocking reads (W-R and W-W)
DSn: Dynamically Scheduled, window size n

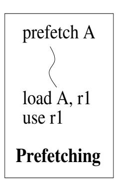
- Latency of reads can be hidden
- Currently studying compiler scheduling to exploit R-R,R-W overlap

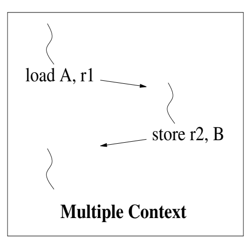
Performance Summary

- Relaxed models effective in hiding memory latency:
  - simple processor, lockup-free cache: 1.1 - 1.4x
  - more aggressive processor: 1.5 - 2.1x
- Gains increase with more processors and higher latency
- Recent results - Zucker and Baer 92

Interaction with Other Latency Hiding Techniques

- Prefetching
  - software controlled
- Multiple contexts
  - switch on long-latency operations

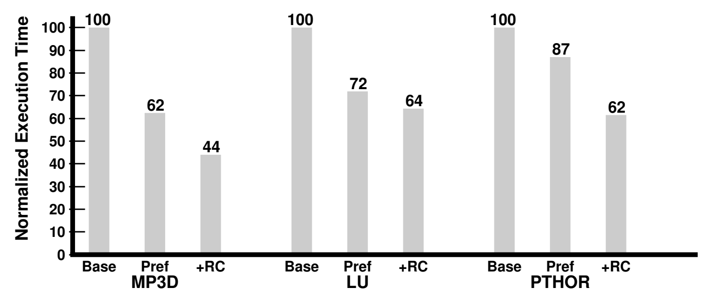




- Conventional processor (not dynamically scheduled)

Interaction with Prefetching

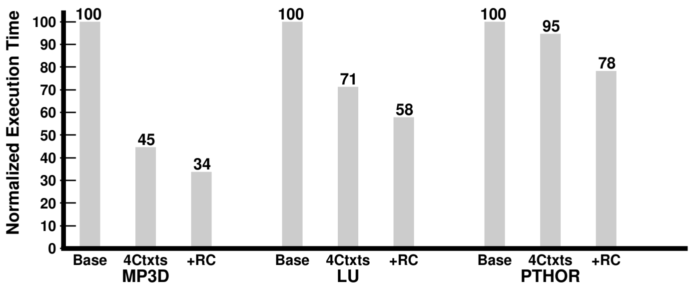
- Prefetching both reads and writes



- Release consistency fully hides remaining write latency

Interaction with Multiple Contexts

• Four contexts, switch latency of 4 cycles



• Write misses no longer require a context switch

Summary of Interaction with Other Techniques

• Release consistency complements prefetching and multiple contexts

- gains over prefetching: 1.1 - 1.4x

- gains over multiple contexts: 1.2 - 1.3x

- lockup-free caches common requirement

Other Gains from Relaxed Models

• Common compiler optimizations require reordering of accesses
  - e.g., register allocation, code motion, loop transformation

• Sequential consistency disallows reordering of shared accesses

• What model is best for compiler optimizations?
  - intermediate models (e.g. PC) not flexible enough
  - weak ordering and release consistency only models that work

Other Gains (continued)

• Software distributed shared-memory

- Munin, Lazy release consistency - Carter 91, Keleher 92

- Entry consistency - Bershad et al. 91

Conclusions

• Relaxed models

- substantial performance gains in hardware and software
- simple abstraction for programmers

• Relaxed models are becoming universal

- Sparc V9, DEC Alpha, Power PC, ...