# Background for Debate on Memory Consistency Models

CS 258, Spring 99
David E. Culler
Computer Science Division
U.C. Berkeley

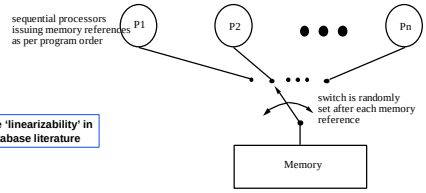
## Memory Consistency Model

- for a SAS specifies constraints on the order in which memory operations (to the same or different locations) can appear to execute with respect to one another,
- enabling programmers to reason about the behavior and correctness of their programs.
- fewer possible reorderings => more intuitive
- more possible reorderings => allows for more performance optimization
  - 'fast but wrong' ?



## Multiprogrammed Uniprocessor Mem. Model

- A MP system 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, and the operations of each individual processor appear in this sequence in the order specified by its program (Lamport)




like 'linearizability' in database literature



## Reasoning with Sequential Consistency

initial: A, flag, x, y == 0

| p1 | p2 |
|---|---|
| (a) A := 1; | (c) x := flag; |
| (b) flag := 1; | (d) y := A |

- program order: (a) ® (b) and (c) ® (d) "preceeds"
- claim: (x,y) == (1,0) cannot occur
  - x == 1 => (b) ® (c)
  - y == 0 => (d) ® (a)
  - thus, (a) ® (b) ® (c) ® (d) ® (a)
  - so (a) ® (a)



## Then again, . . .

initial: A, flag, x, y == 0

| p1 | p2 |
|---|---|
| (a) A := 1; | (c) x := flag; |
| B := 3.1415 | |
| C := 2.78 | |
| (b) flag := 1; | (d) y := A+B+C |

- Many variables are <u>not</u> used to effect the flow of control, but only to shared data
  - synchronizing variables
  - non-synchronizing variables



## Lamport's Requirement for SC

- Each processor <u>issues</u> memory requests in the order specified by its program.
- Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.
- How much overlap is possible?
  - non-memory operations ?
  - memory operations ?
- Assumes stores execute *atomically*
  - newly written value becomes visible to all processors at the same time
    - » inserted into FIFO queue
  - not so with caches and general interconnect

## Requirements for SC (Dubois & Scheurich)

- Each processor issues memory requests in the order specified by the program.
- After a store operation is issued, the issuing processor should wait for the store to complete before issuing its next operation.
- After a load operation is issued, the issuing processor should wait for the load to complete, and for the store whose value is being returned by the load to complete, before issuing its next operation.
- the last point ensures that stores appear atomic to loads
  - note, in an invalidation-based protocol, if a processor has a copy of a block in the dirty state, then a store to the block can complete immediately, since no other processor could access an older value

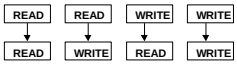


## Architecture Implications


- need write completion for atomicity and access ordering
  - w/o caches, ack writes
  - w/ caches, ack all invalidates
- atomicity
  - delay access to new value till all inv. are acked
- access ordering
  - delay each access till previous completes



## Summary of Sequential Consistency




- Maintain order between shared access in each thread
  - reads or writes wait for previous reads or writes to complete



## Do we really need SC?

- Programmer needs a model to reason with
  - not a different model for each machine

=> Define "correct" as same results as sequential consistency

- Many programs execute correctly even without "strong" ordering

initial: A, flag, x, y == 0

```
p1                  p2
A   := 1;
B   := 3.1415
unlock (L)          lock (L)
                    ... = A;
                    ... = B;
```

- explicit synch operations order key accesses



## Does SC eliminate synchronization?

- No, still need critical sections, barriers, events
  - insert element into a doubly-linked list
  - generation of independent portions of an array
- only ensures interleaving semantics of individual memory operations



## Is SC hardware enough?

- No, Compiler can violate ordering constraints
  - Register allocation to eliminate memory accesses
  - Common subexpression elimination
  - Instruction reordering
  - Software Pipelining

| P1 | P2 | P1 | P2 |
|---|---|---|---|
| B=0 | A=0 | r1=0 | r2=0 |
| A=1 | B=1 | A=1 | B=1 |
| u=B | v=A | u=r1 | v=r2 |
| | | B=r1 | A=r2 |
| (u,v)=(0,0) disallowed under SC | | may occur here | |

- Unfortunately, programming languages and compilers are largely oblivious to memory consistency models
  - languages that take a clear stand, such as HPF too restrictive

## What orderings are essential?

```
initial: A, flag, x, y == 0

p1                  p2
A   := 1;
B   := 3.1415
unlock (L)          lock (L)
                    ... = A;
                    ... = B;
```

- Stores to A and B must complete before unlock
- Loads to A and B must be performed after lock



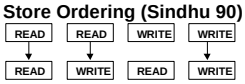
## How do we exploit this?

- Difficult to automatically determine orders that are not necessary
- Relaxed Models:
  - hardware centric: specify orders maintained (or not) by hardware
  - software centric: specify methodology for writing "safe" programs
- All reasonable consistency models retain program order as seen from each processor
  - i.e., dependence order
  - purely sequential code should not break!

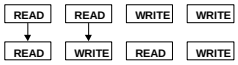


## Hardware Centric Models

- Processor Consistency (Goodman 89)
- Total Store Ordering (Sindhu 90)




- Partial Store Ordering (Sindhu 90)




- Causal Memory (Hutto 90)
- Weak Ordering (Dubois 86)



## Properly Synchronized Programs

- All synchronization operations explicitly identified
- All data accesses ordered though synchronizations
  - no data races!

=> Compiler generated programs from structured high-level parallel languages

=> Structured programming in explicit thread code



## Complete Relaxed Consistency Model

- System specification
  - what program orders among mem operations are preserved
  - what mechanisms are provided to enforce order explicitly, when desired
- Programmer's interface
  - what program annotations are available
  - what 'rules' must be followed to maintain the illusion of SC
- Translation mechanism



## Relaxing write-to-read (PC, TSO)

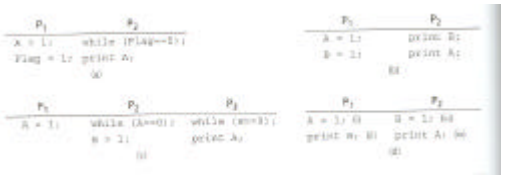
- Why?
  - write-miss in write buffer, later reads hit, maybe even bypass write
- Many common idioms still work

```
initial: A, flag, x, y == 0

p1                  p2
(a) A   := 1;       (c) while (flag ==0) {}
(b) flag := 1;      (d) y := A
```

  - write to flag not visible till previous writes visible
- Ex: Sequent Balance, Encore Multimax, vax 8800, SparcCenter, SGI Challenge, Pentium-Pro

## Detecting weakness wrt SC



- Different results
  - a, b: same for SC, TSO, PC
  - c: PC allows A=0 --- no write atomicity
  - d: TSO and PC allow A=B=0
- Mechanism
  - Sparc V9 provides MEMBAR



## Relaxing write-to-read and write-to-write (PSO)

- Why?
  - write-buffer merging
  - multiple overlapping writes
  - retire out of completion order
- But, even simple use of flags breaks
- Sparc V9 allows write-write membar
- Sparc V8 stbar

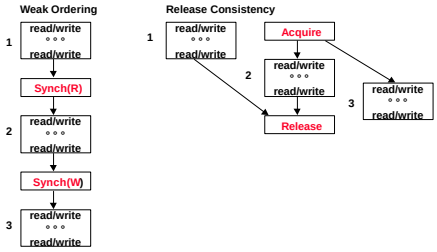


## Relaxing all orders

- Retain control and data dependences within each thread
- Why?
  - allow multiple, overlapping read operations
  - it is what most sequential compilers give you on multithreaded code!
- Weak ordering
  - synchronization operations wait for all previous mem ops to complete
  - arbitrary completion ordering between
- Release Consistency
  - acquire: read operation to gain access to set of operations or variables
  - release: write operation to grant access to others
  - acquire must before following accesses
  - release must wait for preceeding accesses to complete

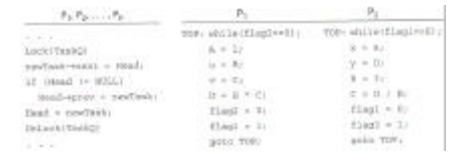


## Preserved Orderings






## Examples





## Programmer's Interface

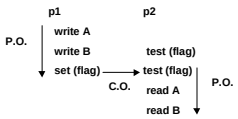
- weak ordering allows programmer to reason in terms of SC, as long as programs are 'data race free'
- release consistency allows programmer to reason in terms of SC for "properly labeled programs"
  - lock is acquire
  - unlock is release
  - barrier is both
  - ok if no synchronization conveyed through ordinary variables

## Identifying Synch events

- two memory operation in different threads conflict is they access same location and one is write
- two conflicting operations compete if one may follow the other in a SC execution with no intervening memory operations on shared data
- a parallel program is synchronized if all competing memory operations have been labeled as synchronization operations
  - perhaps differentiated into acquire and release
- allows programmer to reason in terms of SC, rather than underlying potential reorderings



## Example




- Accesses to flag are competing
  - they constitute a Data Race
  - two conflicting accesses in different threads not ordered by intervening accesses
- Accesses to A (or B) conflict, but do not compete
  - as long as accesses to flag are labeled as synchronizing



## How should programs be labeled?

- Data parallel statements ala HPF
- Library routines
- Variable attributes
- Operators



## Summary of Programmer Model

- Contract between programmer and system:
  - programmer provides synchronized programs
  - system provides effective "sequential consistency" with more room for optimization
- Allows portability over a 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



## Interplay of Micro and multi processor design

- Multiprocessors tend to inherit consistency model from their microprocessor
  - MIPS R10000 -> SGI Origin: SC
  - PPro -> NUMA-Q: PC
  - Sparc: TSO, PSO, RMO
- Can weaken model or strengthen it
- As micros get better at speculation and reordering it is easier to provide SC without as severe performance penalties
  - speculative execution
  - speculative loads
  - write-completion (precise interrupts)



## Questions

- What about larger units of coherence?
  - page-based shared virtual memory
- What happens as latency increases? BW?
- What happens as processors become more sophisticated? Multiple processors on a chip?
- What path should programming languages follow?
  - Java has threads, what's the consistency model?
- How is SC different from transactions?