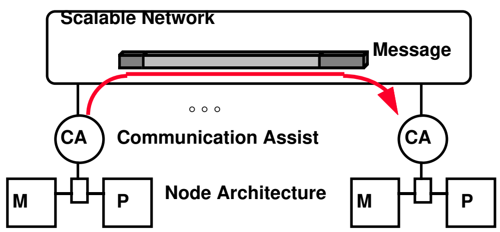
Large-Scale Shared Address Space Multiprocessors

Recap: Dedicated Communication Assist Proc.

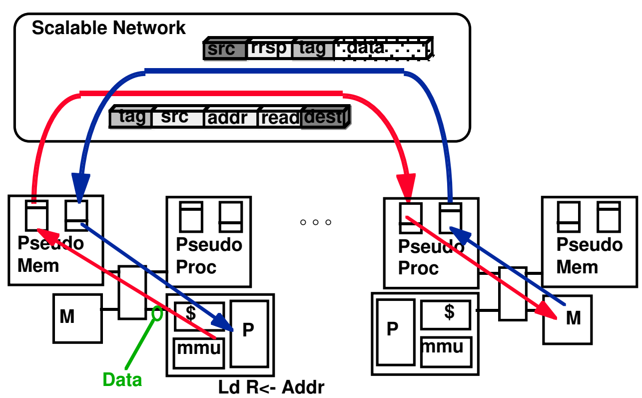


• one: Physical bit stream (blind DMA)	nCUBE, iPSC, . . .
• User/System (ports, handlers)	CM-5, *T, J-Machine, . . .
• Remote virtual address	Paragon, Meiko CS-2
• Global physical address	RP3, BBN, T3D
– **Proc + Memory controller**	
⇒ Cache-to-cache	Dash, KSR, Flash
– **Cache controller**	

Observations

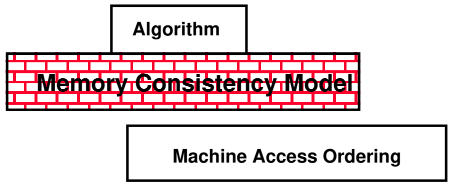
- Few, simple communication operations
  - **can dedicate specialized hardware to the task**
  - **performance**
  - **committment**
- Well-defined semantics
  - **can optimize within the illusion**

Caching Remote Data



- Natural to fetch and cache entire line
- No broadcast and match (snoop)
- Need to keep a record of remotely cached lines
  - **on write, explicitly invalidate other copies of line**

Coherence is not enough



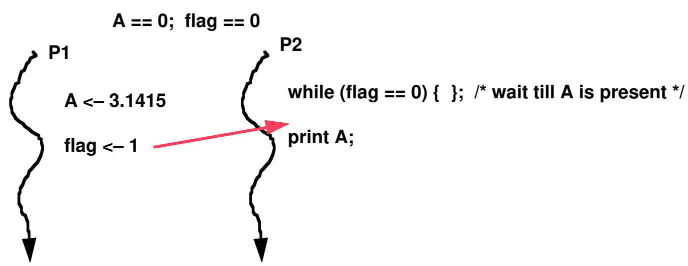
- *memory consistency model* for a shared-memory MP formally specifies how memory operations of a program should appear to execute to the programmer
  - **given a load instruction issued by a processor, it specifies what are the legal values that may be returned for that load given the past execution history of that program**
  - **memory system should return the “latest” value written to that location (???)**
    - » **asynchrony**
    - » **multiple perspectives**
  - **stores to different locations ??????**

Uniprocessor Memory Model

- memory operations appear to execute one at a time, in the order specified by the program (program order)
  - **loads return latest value written to that location**
  - **store binds the value that will be returned by subsequent loads until the next store to that location**
  - **appearance preserved even with caches, write-buffers, interleaved memories, pipelining, out-of-order execution, . . .**
- Program instruction sequence (completely) defines control
- Memory system preserves dependences relative to this seq.
  - **data dependence**
  - **anti-dependence (reuse of storage)**
  - **output dependence**

```
store A
store B
load C
load A
load B
```

Multiple Threads of Control

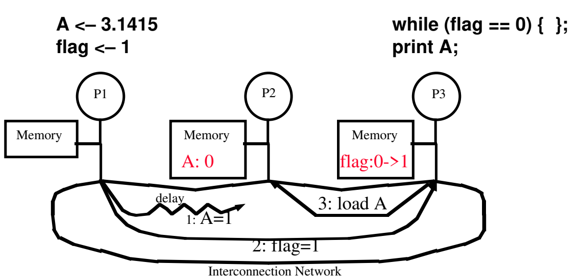


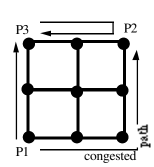
- Memory operations also used to effect control
  - **instruction sequencing among threads**
- Most of what we traditionally do to make the hardware fast breaks the programmer’s intuitive model of what a memory should do.

Unsafe reordering

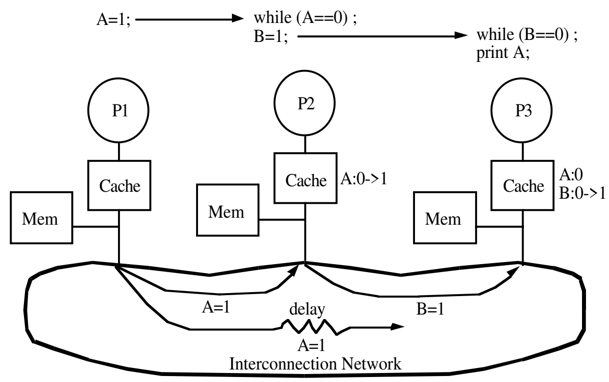
Is it enough to have:

- writes executed in order
- no caches
- point-to-point ordering in the network



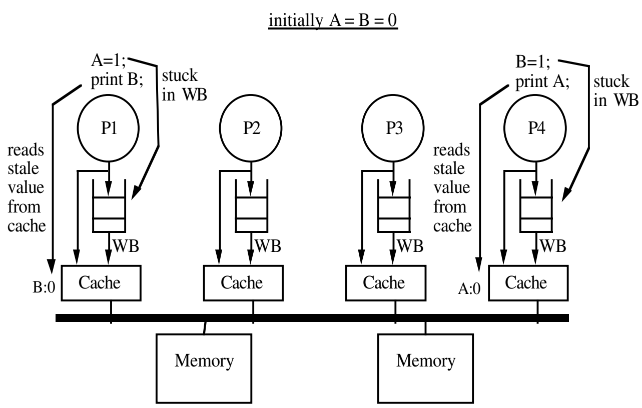


Caches add complications



- Transitive dependence is lost

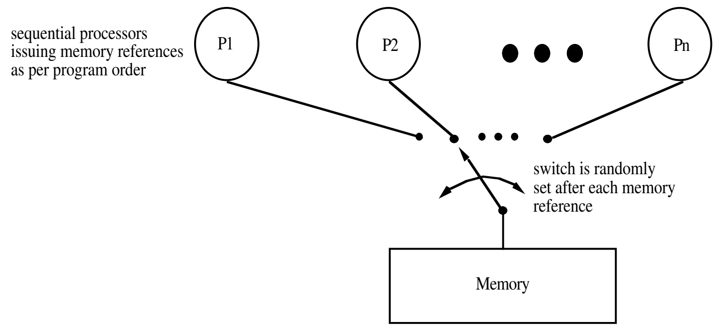
Write-buffers even with CC bus



- Ordered interconnect is not enough

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)



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 not used to effect the flow of control, but only to shared data
  - **synchronizing variables**
  - **non-synchronizing variables**

Lamport's Requirement for SC

- Each processor issues 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**

Definitions

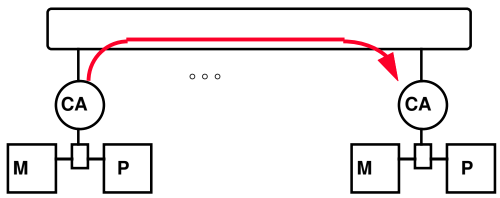
- a load/store is issued when it leaves the processor's internal environment and is in transit in the memory system
  - **the operation becomes potentially visible to other processors**
- a load is complete when its return value is bound
  - **stores issued subsequently by other processors can no longer affect the value returned**
- a store is complete when all other copies of that word have been brought up-to-date
  - **main memory is updated and all cached copies are invalidated or updated**

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
  - **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



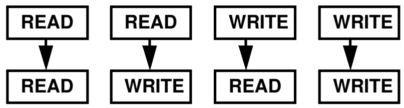
- need write completion for atomicity and access ordering
  - **w/o caches, ack writes**
  - **w/ caches, ack all invalidates**
- atomicity
  - **delay access to new value till all inv. are acked**
- access ordering
  - **delay each access till previous completes**

Implications for compilers

- Register allocation to eliminate memory accesses ???
- Common subexpression elimination ???
- Instruction reordering ???
- Software Pipelining ???

- also S.C. does not eliminate the need for synchronization (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

Summary of Sequential Consistency



- Maintain order between shared access in each thread
- Strong ordering of memory accesses in hardware

Method in the madness

- Programmer needs a model to reason with
  - **not a different model for each machine**
- Define “correct” as same results as sequential consistency
- Most 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;
```

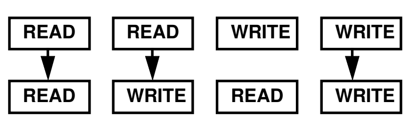
- Stores to A and B must complete before unlock
- Loads to A and B must be performed after lock

How do we exploit this?

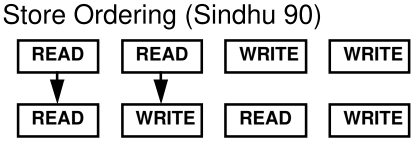
- Difficult to automatically determine orders that are not necessary
- Relaxed Models:
  - **hardware centric: specify orders maintained (or not) by hardware**
  - **software centric: specify methodology for writing “safe” programs**

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

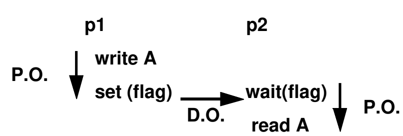
- All synchronization operations explicitly identified
- All data accesses ordered though synchronizations

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

=> Structured programming in explicit thread code

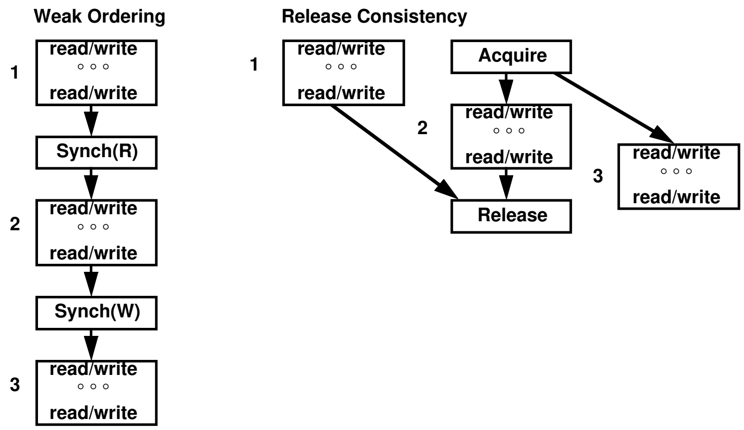
Data Races

- Two accesses <u>conflict</u> if they access the same location and at least one is a store
- Accessed may be ordered by program order or dependence order



- Data Race: two conflicting accesses in different threads not ordered by intervening accesses

Implementation Can Optimize for Synch. Progs



Summary of Programmer Model

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