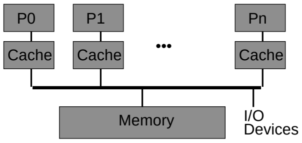
Shared Memory Multiprocessors

By constructing a standard sequential processor and its memory system intelligently, i.e. adopting a slightly more general design, a multiprocessor element can be constructed

Basic Architecture of SMP

Buses are good news and bad news

- The (memory) bus is a point all processors can "see" and thus be informed of what is happening
- A bus is serially used, so can be a bottleneck

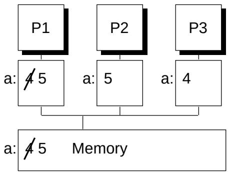


Cache Coherency -- The Problem

- Independent processors modifying shared locations can change values without other processors being aware of it

P1 reads a into cache
P3 reads a into cache
P1 writes a to 5, and writes through to main memory, correcting it
-- P3 has stale data --
P2 reads a into cache

Memory is arbiter of present value ... move it closer to processors



Cache Coherency, The Goal

- A multiprocessor memory system is coherent if for every location there exists a serial order for the operations on that location consistent with the results of the execution such that ...
  - The subsequence of operations issued by any processor are in the order issued
  - The value returned by each read is the value written by the last write in serial order

Implied property of coherency ...
Write Serialization -- all writes to a location are seen in the same order by all processes

Snooping To Solve Coherency

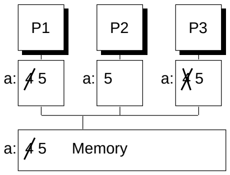
- The cache controllers can "snoop" on the bus, meaning that they watch the events on the bus even if they do not issue them, noting any action applied to cache lines they hold
- There are two possible actions when a memory location held by processor A is changed by processor B
  - Invalidate -- mark local copy as invalid
  - Update -- make the same change B made

The unit of cache coherency is a cache line or block

Snooping At Work

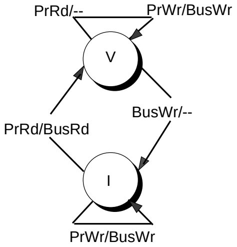
By snooping the cache controller for processor P3 can take action in response to P1's write

P1 reads a into cache
P3 reads a into cache
P1 writes a to 5, and writes through to main memory, correcting it
P3 sees WT, invalidates or updates
P2 reads a into cache



Write-through Coherency

- State diagrams show the protocol ...



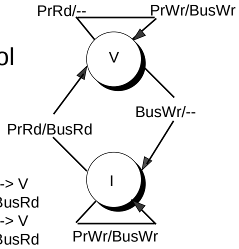
States on cache line ...
V is valid
I is invalid

Transitions ...
Red is processor initiated
Blue is bus initiated

Labeling A/B ...
If A is observed
Then transaction B
is generated

Applying The WT Protocol

Consider the transitions

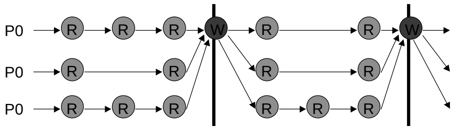


P1 reads a into cache
P3 reads a into cache
P1 writes a to 5, and writes through to main memory, correcting it
P3 sees WT, invalidates
P2 reads a into cache

P1: I ---> V
PrRd/BusRd
P3: I ---> V
PrRd/BusRd
P1: V ---> V
PrWr/BusWr
P3: V ---> I
BusWr/--
P2: I ---> V
PrRd/BusRd

Partial Order On Memory Operations

Write bus transactions define a global sequence of events between which processors can read ... any total order produced by interleaving



Memory Consistency

- What should it mean for processors to see a consistent view of memory?
- Coherency is too weak because it only requires ordering with respect to individual locations, but there are other ways of binding values together

```
   -- a and flag are 0 initially --
P0                  P1
a    := 1;          while (flag != 1) ; -- spin
flag := 1;          print a;
```

SC -- Sequential Consistency

Lamport: A multiprocessor is sequentially consistent if the result of any execution is the same as some sequential order, and within any processor, the operations are executed in program order

```
   -- a and b are 0 initially --
P0                  P1
a := 1;             print b;
b := 2;             print a;
```

Possible Results

a	b
0	0
1	0
0	2
1	2

Write Atomicity

- Write atomicity says that all writes to a location should appear to all processors to have occurred in the same order

```
               -- a and b are 0 initially --
P0             P1                  P2
a := 1;        while a != 1 do;    while b != 1 do;
               b := 1;             print a;
```

Sufficient Conditions For SC

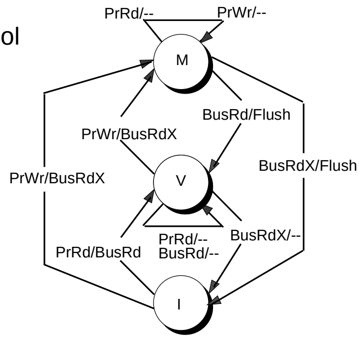
- Three conditions suffice for SC
  - Memory requests are issued in program order
  - Wait for writes to complete
  - Wait for reads to complete and the write filling the read to complete

```
                -- a and b are 0 initially --
P0              P1                 P2
a := 1;         while a != 1 do;   while b != 1 do;
                b := 1;            print a;
```

Basic Write-back Snoopy Cache Designs

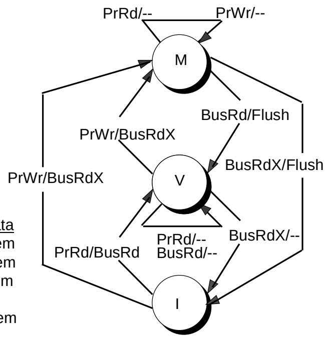
- Write-back protocols are more complex than write-through because modified data remains in the cache
- Introduce more cache states
  - Modified, or dirty, the value differs from memory
  - Exclusive, no other cache has this value
- We consider three
  - MSI, an invalidation protocol
  - MESI, an invalidation protocol
  - Dragon, an update protocol

MSI Protocol



MSI In Action

Proc Action	P0	P1	P2	Bus	Data
P0: r a	S	-	-	BRd	Mem
P2: r a	S	-	S	BRd	Mem
P2:w a	I	-	M	BRdx	Mem
P0: r a	S	-	S	BRd	P2
P1: r a	S	S	S	BRd	Mem



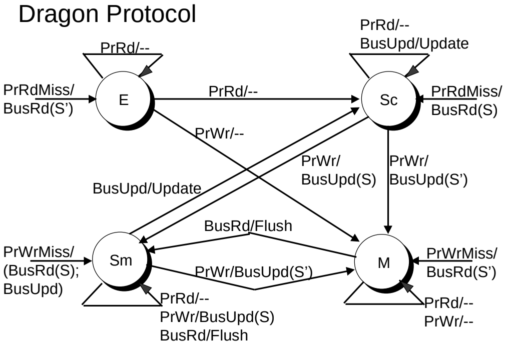
MESI (Illinois) Protocol

- The problem with MSI is that 2 bus transactions are needed just to load and update a value even with no one is sharing
- Add a new state to get 4
  - M = modified or dirty, value differs from memory
  - E = exclusive, clean, one cached copy
  - S = shared, multiple cached copies
  - I = invalid

Dragon -- An Update Protocol

- The caches are the valid memory contents -- memory is changed only when a line is needed
- Introduce Shared clean (Sc) and Shared Modified (Sm) states, dump Invalid
- Need to introduce the concept of Read Miss and Write Miss
- Add a “shared” line to the bus
- The basic idea: Keep all lines of all caches current -- note that updates will update modified word only

Dragon Protocol



PrRd/--

E

PrRdMiss/
BusRd(S')

PrRd/--

PrRd/--
BusUpd/Update

Sc

PrRdMiss/
BusRd(S)

PrWr/--

BusUpd/Update

PrWr/
BusUpd(S)

PrWr/
BusUpd(S')

BusRd/Flush

PrWrMiss/
(BusRd(S);
BusUpd)

Sm

PrWr/BusUpd(S')

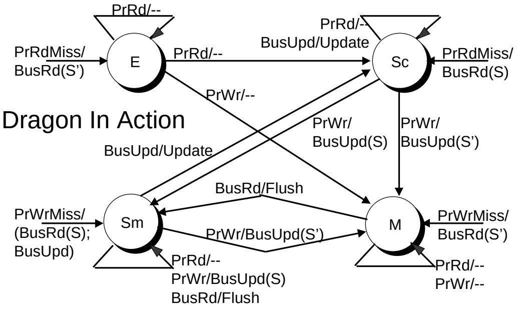
M

PrWrMiss/
BusRd(S')

PrRd/--
PrWr/BusUpd(S)
BusRd/Flush

PrRd/--
PrWr/--

Dragon In Action



PrRd/--

PrRdMiss/
BusRd(S')

E

PrRd/--

PrRd/--
BusUpd/Update

Sc

PrRdMiss/
BusRd(S)

PrWr/--

BusUpd/Update

PrWr/
BusUpd(S)

PrWr/
BusUpd(S')

BusRd/Flush

PrWrMiss/
(BusRd(S);
BusUpd)

Sm

PrWr/BusUpd(S')

M

PrWrMiss/
BusRd(S')

PrRd/--
PrWr/BusUpd(S)
BusRd/Flush

PrRd/--
PrWr/--

Proc Action	P0	P1	P2	Bus	Data
P0: r a	E	-	-	BRd	Mem
P2: r a	Sc	-	Sc	BRd	Mem
P2:w a	Sc	-	Sm	BUpd	P2

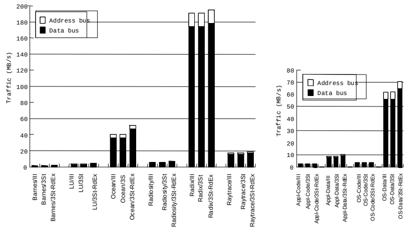
Proc Action	P0	P1	P2	Bus	Data
P0: r a	Sc	-	Sm	null	-
P1: r a	Sc	Sc	Sm	BRd	P2

Assessing trade-offs

- You cannot design a protocol and reason through how it will work, and probably not even whether it is correct ...
- Many criteria for success
  - Good use of bandwidth
  - Rapid response
- Performance must be relative to some computation ... what's typical?

Protocol Optimizations: Worth It?

Effect of E state, and of BusUpgr instead of BusRdX

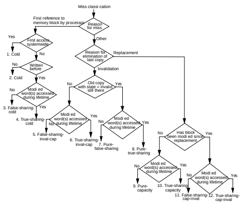


MSI vs MESI … little difference; Upgrade helps some

Graph from Text

Caching Properties

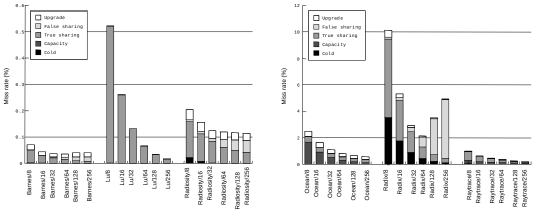
- Cache misses have long been categorized …
  - Compulsory
  - Capacity
  - Conflict
- Add "Coherence"
- Sharing in 2 forms
  - True
  - False


Miss classi cation
First reference to memory block by processor
Reason for miss
Yes
First access systemwide
1. Cold
No
Other
No
Written before
2. Cold
Yes
Reason for elimination of last copy
Replacement
Invalidation
No
Modi ed word(s) accessed during lifetime
3. False-sharing-cold
Yes
4. True-sharing-cold
No
Old copy with state = invalid still there
Yes
No
Modi ed word(s) accessed during lifetime
Yes
5. False-sharing-inval-cap
6. True-sharing-inval-cap
No
Modi ed word(s) accessed during lifetime
Yes
7. Pure-false-sharing
8. Pure-true-sharing
No
Has block been modi ed since replacement
Yes
No
Modi ed word(s) accessed during lifetime
Yes
9. Pure-capacity
10. True-sharing-capacity
No
Modi ed word(s) accessed during lifetime
Yes
11. False-sharing-cap-inval
12. True-sharing-cap-inval

Graph from Text

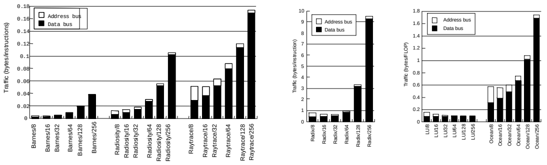
Cache-block size

- Several features illustrated



Graph from Text

Block Size Affects Traffic



- Contention increases

Graph from Text

Update vs Invalidate

- Intuition --

  If use implies continued to use, and writes between use are few, update should do better

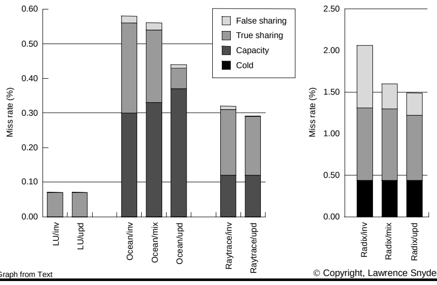
  - e.g. producer-consumer pattern

  If use implies 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

Update vs Invalidation

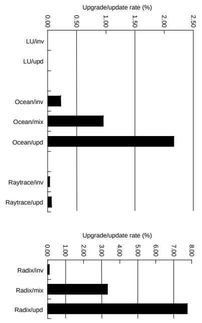
- Much coherence: updates help
- Much capacity updates hurt



Graph from Text

There’s More To Story

- Bus traffic is huge
- A single processor tends to write a lot before other proc resds
- Many bus updates vs one invalidate!



Graph from Text

Synchronization

- A long and glorious past … ‘67
- A huge time cost in parallel programs
- Though studied intensively it is still not really solved

  The problem: Processes must share information, but its integrity must be preserved
- Some hardware assist is essential in order to achieve atomicity
- User say, “Just give me primitives that work”

Simple Software Lock

```
lock:       ld    R1,loc  -- get fresh value
            cmp   loc, #0 -- test if it changed
            bnz   R1,lock -- spin if loc ~ free?
            st    loc, #1 -- its free, set it
            ret
and
unlock:     st loc,#0     -- clear setting
            ret
```

The code seems simple enough, but consider various interleavings

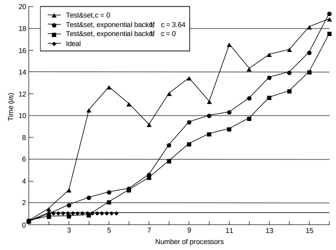
Test&Set Is A Simple Solution

- `Test_and_set  R1, Loc` fetches `Loc`'s value, and sets it to 1, returning value to `R1`
- Consider its operation

```
lock:       t&s   R1,Loc  -- atomically set
            bnz   R1,loc  -- spin if loc ~ free?
            ret
and
unlock:     st Loc,#0     -- clear setting
            ret
```

Performance Of a Lock

- lock; delay (c) ; unlock on SGI Challenge



Graph from Text

Performance Goals of Locks

- Locks affect performance -- critical aspects
  - Low Latency
  - Low Traffic
  - Scalability
  - Low Storage Cost
  - Fairness

Improved Hardware Primitives

Seek a basic primitive suitable for range of cases

- *Load-Locked* (or *Load-Linked*), *Store-Conditional*
- LL reads location into a register
- Follow with arbitrary instructions to manipulate value
- SC tries to store back to location if and only if no other processor has written to the variable since this processor's LL
  - If SC succeeds, all three steps happened "atomically"
  - If fail, don't write or generate invalidations (must retry LL)
  - Success indicated by condition codes

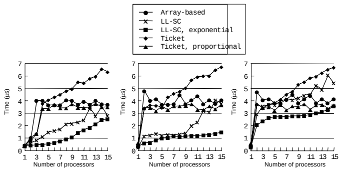
Sample Use of LL-SC

```
lock: ll    R1, Loc     -- Load-lock Loc to R1
      bnz   R1, lock    -- Spin if Loc locked
      sc    Loc, R2     -- Cond'ly store R2 in Loc
      beqz  lock        -- If failed, repeat
      ret
and
unlock: st Loc, #0      -- Clear location
        ret
```

- Many processes can do an LL at once, but only the first to the SC wins

Performance

`lock; delay(c); unlock; delay(d)` on SGI Challenge

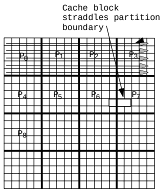


(a) Null ($c = 0$, $d = 0$)   (b) Critical-section (c = 3.64 μs, d = 0)   (c) Delay (c = 3.64 μs, d = 1.29 μs)

Graph from Text

Software Implications

- Blocked allocation of 2D array
- References straddling cache lines loses on
  - Fragmentation
  - False Sharing



(a) Two-dimensional array

Graph from Text