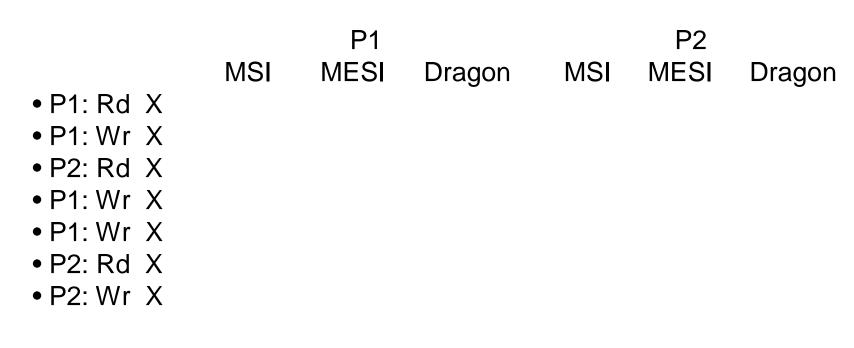
• Topics: evaluating coherence, synchronization primitives

1

Example



Total transfers:

Evaluating Coherence Protocols

- There is no substitute for detailed simulation high communication need not imply poor performance if the communication is off the critical path – for example, an update protocol almost always consumes more bandwidth, but can often yield better performance
- An easy (though, not entirely reliable) metric simulate cache accesses and compute state transitions – each state transition corresponds to a fixed amount of interconnect traffic

State Transitions

То) NP	I	E	S			М			
From										
NP	0	0	1.25	0	0.96		1.68	State transitions per 1000 data memory references		
I	0.64	0	0	1.87		0	0.002			
E	0.20	0	14.0	0.02			1.00	me	for Ocean	
S	0.42	2.5	0	134.7			2.24			
М	2.63	0.002	0		2.3	8	343.6			
То	NP	I	E		S		М			
From									Bus actions	
NP			BusRd		BusR		BusRdX		for each state	
I			BusRd		BusRd		BusRdX		transition	
E										
S			Not possible				BusUpgr			
М	BusWB	BusWB	Not possible		BusWB				4	

- Coherence misses: cache misses caused by sharing of data blocks – true (two different processes access the same word) and false (processes access different words in the same cache line)
- False coherence misses are zero if the block size equals the word size
- An upgrade from S to M is a new type of "cache miss" as it generates (inexpensive) bus traffic

Block Size

- For most programs, a larger block size increases the number of false coherence misses, but significantly reduces most other types of misses (because of locality)
 – a very large block size will finally increase conflict misses
- Large block sizes usually result in high bandwidth needs in spite of the lower miss rate
- Alleviating false sharing drawbacks of a large block size:
 - maintain state information at a finer granularity (in other words, prefetch multiple blocks on a miss)
 - delay write invalidations
 - reorganize data structures and decomposition

Update-Invalidate Trade-Offs

- The best performing protocol is a function of sharing patterns – are the sharers likely to read the newly updated value? Examples: locks, barriers, diff
- Each variable in the program has a different sharing pattern – what can we do?
- Implement both protocols in hardware let the programmer/hw select the protocol for each page/block
- For example: in the Dragon update protocol, maintain a counter for each block – an access sets the counter to MAX, while an update decrements it – if the counter reaches 0, the block is evicted

- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write
- Atomic exchange: swap contents of register and memory
- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory
- lock: t&s register, location
 bnz register, lock
 CS
 st location, #0

Improving Lock Algorithms

- The basic lock implementation is inefficient because the waiting process is constantly attempting writes → heavy invalidate traffic
- Test & Set with exponential back-off: if you fail again, double your wait time and try again
- Test & Test & Set: read the value, if it has not changed, don't bother doing the test&set – heavy bus traffic only when the lock is released
- Different implementations trade-off one of these lock properties: latency, traffic, scalability, storage, fairness

Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility
- LL: read a value and update a table indicating you have read this address, then perform any amount of computation
- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL
- SC implementations may not generate bus traffic if the SC fails hence, more efficient than test&test&set

Load-Linked and Store Conditional

Iockit:LLR2, 0(R1); load linked, generates no coherence trafficBNEZR2, lockit; not available, keep spinningDADDUI R2, R0, #1; put value 1 in R2SCR2, 0(R1); store-conditional succeeds if no one; updated the lock since the last LLBEQZR2, lockit; confirm that SC succeeded, else keep trying

Further Reducing Bandwidth Needs

- Even with LL-SC, heavy traffic is generated on a lock release and there are no fairness guarantees
- Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable – is this really better than the LL-SC implementation?
- Array-Based lock: instead of using a "now-serving" variable, use a "now-serving" array and each process waits on a different variable – fair, low latency, low bandwidth, high scalability, but higher storage

Barriers

- Barriers require each process to execute a lock and unlock to increment the counter and then spin on a shared variable
- If multiple barriers use the same variable, deadlock can arise because some process may not have left the earlier barrier – sense-reversing barriers can solve this problem
- A tree can be employed to reduce contention for the lock and shared variable
- When one process issues a read request, other processes can snoop and update their invalid entries 13

Barrier Implementation

```
LOCK(bar.lock);

if (bar.counter == 0)

bar.flag = 0;

mycount = bar.counter++;

UNLOCK(bar.lock);

if (mycount == p) {

bar.counter = 0;

bar.flag = 1;

}

else

while (bar.flag == 0) { };
```

Sense-Reversing Barrier Implementation

```
local_sense = !(local_sense);
LOCK(bar.lock);
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
  bar.counter = 0;
  bar.flag = local_sense;
}
else {
  while (bar.flag != local_sense) { };
}
```

Title

• Bullet